util.py (original)

```
# util.py
# -------
# Licensing Information: Please do not distribute or publish solutions to this
# project. You are free to use and extend these projects for educational
# purposes. The Pacman AI projects were developed at UC Berkeley, primarily by
# John DeNero (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# For more info, see http://inst.eecs.berkeley.edu/~cs188/sp09/pacman.html

import sys
import inspect
import heapq, random


"""
 Data structures useful for implementing SearchAgents
"""

class Stack:
  "A container with a last-in-first-out (LIFO) queuing policy."
  def __init__(self):
    self.list = []

  def push(self,item):
    "Push 'item' onto the stack"
    self.list.append(item)

  def pop(self):
    "Pop the most recently pushed item from the stack"
    return self.list.pop()

  def isEmpty(self):
    "Returns true if the stack is empty"
    return len(self.list) == 0

class Queue:
  "A container with a first-in-first-out (FIFO) queuing policy."
  def __init__(self):
    self.list = []

  def push(self,item):
    "Enqueue the 'item' into the queue"
    self.list.insert(0,item)

  def pop(self):
    """
      Dequeue the earliest enqueued item still in the queue. This
      operation removes the item from the queue.
    """
    return self.list.pop()

  def isEmpty(self):
    "Returns true if the queue is empty"
    return len(self.list) == 0

class PriorityQueue:
  """
    Implements a priority queue data structure. Each inserted item
    has a priority associated with it and the client is usually interested
    in quick retrieval of the lowest-priority item in the queue. This
    data structure allows O(1) access to the lowest-priority item.

    Note that this PriorityQueue does not allow you to change the priority
    of an item.  However, you may insert the same item multiple times with
    different priorities.
```

```
    """
    def  __init__(self):
      self.heap = []

    def push(self, item, priority):
        pair = (priority,item)
        heapq.heappush(self.heap,pair)

    def pop(self):
        (priority,item) = heapq.heappop(self.heap)
        return item

    def isEmpty(self):
      return len(self.heap) == 0

class PriorityQueueWithFunction(PriorityQueue):
    """
    Implements a priority queue with the same push/pop signature of the
    Queue and the Stack classes. This is designed for drop-in replacement for
    those two classes. The caller has to provide a priority function, which
    extracts each item's priority.
    """
    def  __init__(self, priorityFunction):
      "priorityFunction (item) -> priority"
      self.priorityFunction = priorityFunction      # store the priority function
      PriorityQueue.__init__(self)        # super-class initializer

    def push(self, item):
      "Adds an item to the queue with priority from the priority function"
      PriorityQueue.push(self, item, self.priorityFunction(item))


def manhattanDistance( xy1, xy2 ):
    "Returns the Manhattan distance between points xy1 and xy2"
    return abs( xy1[0] - xy2[0] ) + abs( xy1[1] - xy2[1] )

"""
    Data structures and functions useful for various course projects

    The search project should not need anything below this line.
"""

class Counter(dict):
    """
    A counter keeps track of counts for a set of keys.

    The counter class is an extension of the standard python
    dictionary type.  It is specialized to have number values
    (integers or floats), and includes a handful of additional
    functions to ease the task of counting data.  In particular,
    all keys are defaulted to have value 0.  Using a dictionary:

    a = {}
    print a['test']

    would give an error, while the Counter class analogue:

    >>> a = Counter()
    >>> print a['test']
    0

    returns the default 0 value. Note that to reference a key
    that you know is contained in the counter,
    you can still use the dictionary syntax:

    >>> a = Counter()
    >>> a['test'] = 2
    >>> print a['test']
    2
```

```
This is very useful for counting things without initializing their counts,
see for example:

>>> a['blah'] += 1
>>> print a['blah']
1

The counter also includes additional functionality useful in implementing
the classifiers for this assignment.  Two counters can be added,
subtracted or multiplied together.  See below for details.  They can
also be normalized and their total count and arg max can be extracted.
"""
def __getitem__(self, idx):
  self.setdefault(idx, 0)
  return dict.__getitem__(self, idx)

def incrementAll(self, keys, count):
  """
  Increments all elements of keys by the same count.

  >>> a = Counter()
  >>> a.incrementAll(['one','two', 'three'], 1)
  >>> a['one']
  1
  >>> a['two']
  1
  """
  for key in keys:
    self[key] += count

def argMax(self):
  """
  Returns the key with the highest value.
  """
  if len(self.keys()) == 0: return None
  all = self.items()
  values = [x[1] for x in all]
  maxIndex = values.index(max(values))
  return all[maxIndex][0]

def sortedKeys(self):
  """
  Returns a list of keys sorted by their values.  Keys
  with the highest values will appear first.

  >>> a = Counter()
  >>> a['first'] = -2
  >>> a['second'] = 4
  >>> a['third'] = 1
  >>> a.sortedKeys()
  ['second', 'third', 'first']
  """
  sortedItems = self.items()
  compare = lambda x, y:  sign(y[1] - x[1])
  sortedItems.sort(cmp=compare)
  return [x[0] for x in sortedItems]

def totalCount(self):
  """
  Returns the sum of counts for all keys.
  """
  return sum(self.values())

def normalize(self):
  """
  Edits the counter such that the total count of all
  keys sums to 1.  The ratio of counts for all keys
  will remain the same. Note that normalizing an empty
```

```
    Counter will result in an error.
    """
    total = float(self.totalCount())
    if total == 0: return
    for key in self.keys():
      self[key] = self[key] / total

  def divideAll(self, divisor):
    """
    Divides all counts by divisor
    """
    divisor = float(divisor)
    for key in self:
      self[key] /= divisor

  def copy(self):
    """
    Returns a copy of the counter
    """
    return Counter(dict.copy(self))

  def __mul__(self, y ):
    """
    Multiplying two counters gives the dot product of their vectors where
    each unique label is a vector element.

    >>> a = Counter()
    >>> b = Counter()
    >>> a['first'] = -2
    >>> a['second'] = 4
    >>> b['first'] = 3
    >>> b['second'] = 5
    >>> a['third'] = 1.5
    >>> a['fourth'] = 2.5
    >>> a * b
    14
    """
    sum = 0
    x = self
    if len(x) > len(y):
      x,y = y,x
    for key in x:
      if key not in y:
        continue
      sum += x[key] * y[key]
    return sum

  def __radd__(self, y):
    """
    Adding another counter to a counter increments the current counter
    by the values stored in the second counter.

    >>> a = Counter()
    >>> b = Counter()
    >>> a['first'] = -2
    >>> a['second'] = 4
    >>> b['first'] = 3
    >>> b['third'] = 1
    >>> a += b
    >>> a['first']
    1
    """
    for key, value in y.items():
      self[key] += value

  def __add__( self, y ):
    """
    Adding two counters gives a counter with the union of all keys and
    counts of the second added to counts of the first.
```

```
    >>> a = Counter()
    >>> b = Counter()
    >>> a['first'] = -2
    >>> a['second'] = 4
    >>> b['first'] = 3
    >>> b['third'] = 1
    >>> (a + b)['first']
    1
    """
    addend = Counter()
    for key in self:
      if key in y:
        addend[key] = self[key] + y[key]
      else:
        addend[key] = self[key]
    for key in y:
      if key in self:
        continue
      addend[key] = y[key]
    return addend

  def __sub__( self, y ):
    """
    Subtracting a counter from another gives a counter with the union of all keys and
    counts of the second subtracted from counts of the first.

    >>> a = Counter()
    >>> b = Counter()
    >>> a['first'] = -2
    >>> a['second'] = 4
    >>> b['first'] = 3
    >>> b['third'] = 1
    >>> (a - b)['first']
    -5
    """
    addend = Counter()
    for key in self:
      if key in y:
        addend[key] = self[key] - y[key]
      else:
        addend[key] = self[key]
    for key in y:
      if key in self:
        continue
      addend[key] = -1 * y[key]
    return addend

def raiseNotDefined():
  print "Method not implemented: %s" % inspect.stack()[1][3]
  sys.exit(1)

def normalize(vectorOrCounter):
  """
  normalize a vector or counter by dividing each value by the sum of all values
  """
  normalizedCounter = Counter()
  if type(vectorOrCounter) == type(normalizedCounter):
    counter = vectorOrCounter
    total = float(counter.totalCount())
    if total == 0: return counter
    for key in counter.keys():
      value = counter[key]
      normalizedCounter[key] = value / total
    return normalizedCounter
  else:
    vector = vectorOrCounter
    s = float(sum(vector))
    if s == 0: return vector
```

```
    return [el / s for el in vector]

def nSample(distribution, values, n):
  if sum(distribution) != 1:
    distribution = normalize(distribution)
  rand = [random.random() for i in range(n)]
  rand.sort()
  samples = []
  samplePos, distPos, cdf = 0,0, distribution[0]
  while samplePos < n:
    if rand[samplePos] < cdf:
      samplePos += 1
      samples.append(values[distPos])
    else:
      distPos += 1
      cdf += distribution[distPos]
  return samples

def sample(distribution, values = None):
  if type(distribution) == Counter:
    items = distribution.items()
    distribution = [i[1] for i in items]
    values = [i[0] for i in items]
  if sum(distribution) != 1:
    distribution = normalize(distribution)
  choice = random.random()
  i, total= 0, distribution[0]
  while choice > total:
    i += 1
    total += distribution[i]
  return values[i]

def sampleFromCounter(ctr):
  items = ctr.items()
  return sample([v for k,v in items], [k for k,v in items])

def getProbability(value, distribution, values):
  """
    Gives the probability of a value under a discrete distribution
    defined by (distributions, values).
  """
  total = 0.0
  for prob, val in zip(distribution, values):
    if val == value:
      total += prob
  return total

def flipCoin( p ):
  r = random.random()
  return r < p

def chooseFromDistribution( distribution ):
  "Takes either a counter or a list of (prob, key) pairs and samples"
  if type(distribution) == dict or type(distribution) == Counter:
    return sample(distribution)
  r = random.random()
  base = 0.0
  for prob, element in distribution:
    base += prob
    if r <= base: return element

def nearestPoint( pos ):
  """
  Finds the nearest grid point to a position (discretizes).
  """
  ( current_row, current_col ) = pos

  grid_row = int( current_row + 0.5 )
  grid_col = int( current_col + 0.5 )
```

```
  return ( grid_row, grid_col )

def sign( x ):
  """
  Returns 1 or -1 depending on the sign of x
  """
  if( x >= 0 ):
    return 1
  else:
    return -1

def arrayInvert(array):
  """
  Inverts a matrix stored as a list of lists.
  """
  result = [[] for i in array]
  for outer in array:
    for inner in range(len(outer)):
      result[inner].append(outer[inner])
  return result

def matrixAsList( matrix, value = True ):
  """
  Turns a matrix into a list of coordinates matching the specified value
  """
  rows, cols = len( matrix ), len( matrix[0] )
  cells = []
  for row in range( rows ):
    for col in range( cols ):
      if matrix[row][col] == value:
        cells.append( ( row, col ) )
  return cells

def lookup(name, namespace):
  """
  Get a method or class from any imported module from its name.
  Usage: lookup(functionName, globals())
  """
  dots = name.count('.')
  if dots > 0:
    moduleName, objName = '.'.join(name.split('.')[:-1]), name.split('.')[-1]
    module = __import__(moduleName)
    return getattr(module, objName)
  else:
    modules = [obj for obj in namespace.values() if str(type(obj)) == "<type
'module'>"]
    options = [getattr(module, name) for module in modules if name in dir(module)]
    options += [obj[1] for obj in namespace.items() if obj[0] == name ]
    if len(options) == 1: return options[0]
    if len(options) > 1: raise Exception, 'Name conflict for %s'
    raise Exception, '%s not found as a method or class' % name

def pause():
  """
  Pauses the output stream awaiting user feedback.
  """
  print "<Press enter/return to continue>"
  raw_input()


## code to handle timeouts
import signal
class TimeoutFunctionException(Exception):
    """Exception to raise on a timeout"""
    pass

class TimeoutFunction:

    def __init__(self, function, timeout):
```

```
        "timeout must be at least 1 second. WHY??"
        self.timeout = timeout
        self.function = function

    def handle_timeout(self, signum, frame):
        raise TimeoutFunctionException()

    def __call__(self, *args):
        if not 'SIGALRM' in dir(signal):
            return self.function(*args)
        old = signal.signal(signal.SIGALRM, self.handle_timeout)
        signal.alarm(self.timeout)
        try:
            result = self.function(*args)
        finally:
            signal.signal(signal.SIGALRM, old)
        signal.alarm(0)
        return result
```