~\Desktop\Assessed Lab\utils.py

```python
"""Provides some utilities widely used by other modules"""

import bisect
import collections
import collections.abc
import functools
import heapq
import operator
import os.path
import random
from itertools import chain, combinations
from statistics import mean

import numpy as np


# _____________________________________________________________________________
# Functions on Sequences and Iterables


def sequence(iterable):
    """Converts iterable to sequence, if it is not already one."""
    return iterable if isinstance(iterable, collections.abc.Sequence) else tuple([iterable])


def remove_all(item, seq):
    """Return a copy of seq (or string) with all occurrences of item removed."""
    if isinstance(seq, str):
        return seq.replace(item, '')
    elif isinstance(seq, set):
        rest = seq.copy()
        rest.remove(item)
        return rest
    else:
        return [x for x in seq if x != item]


def unique(seq):
    """Remove duplicate elements from seq. Assumes hashable elements."""
    return list(set(seq))


def count(seq):
    """Count the number of items in sequence that are interpreted as true."""
    return sum(map(bool, seq))


def multimap(items):
    """Given (key, val) pairs, return {key: [val, ....], ...}."""
    result = collections.defaultdict(list)
    for (key, val) in items:
```

```
52          result[key].append(val)
53      return dict(result)
54
55
56  def multimap_items(mmap):
57      """Yield all (key, val) pairs stored in the multimap."""
58      for (key, vals) in mmap.items():
59          for val in vals:
60              yield key, val
61
62
63  def product(numbers):
64      """Return the product of the numbers, e.g. product([2, 3, 10]) == 60"""
65      result = 1
66      for x in numbers:
67          result *= x
68      return result
69
70
71  def first(iterable, default=None):
72      """Return the first element of an iterable; or default."""
73      return next(iter(iterable), default)
74
75
76  def is_in(elt, seq):
77      """Similar to (elt in seq), but compares with 'is', not '=='."""
78      return any(x is elt for x in seq)
79
80
81  def mode(data):
82      """Return the most common data item. If there are ties, return any one of them."""
83      [(item, count)] = collections.Counter(data).most_common(1)
84      return item
85
86
87  def power_set(iterable):
88      """power_set([1,2,3]) --> (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"""
89      s = list(iterable)
90      return list(chain.from_iterable(combinations(s, r) for r in range(len(s) + 1)))[1:]
91
92
93  def extend(s, var, val):
94      """Copy dict s and extend it by setting var to val; return copy."""
95      return {**s, var: val}
96
97
98  def flatten(seqs):
99      return sum(seqs, [])
100
101
102 # _____________________________________________________________________________
103 # argmin and argmax
104
105 identity = lambda x: x
```

```
106
107
108 def argmin_random_tie(seq, key=identity):
109     """Return a minimum element of seq; break ties at random."""
110     return min(shuffled(seq), key=key)
111
112
113 def argmax_random_tie(seq, key=identity):
114     """Return an element with highest fn(seq[i]) score; break ties at random."""
115     return max(shuffled(seq), key=key)
116
117
118 def shuffled(iterable):
119     """Randomly shuffle a copy of iterable."""
120     items = list(iterable)
121     random.shuffle(items)
122     return items
123
124
125 # ____________________________________________________________________________
126 # Statistical and mathematical functions
127
128
129 def histogram(values, mode=0, bin_function=None):
130     """Return a list of (value, count) pairs, summarizing the input values.
131     Sorted by increasing value, or if mode=1, by decreasing count.
132     If bin_function is given, map it over values first."""
133     if bin_function:
134         values = map(bin_function, values)
135
136     bins = {}
137     for val in values:
138         bins[val] = bins.get(val, 0) + 1
139
140     if mode:
141         return sorted(list(bins.items()), key=lambda x: (x[1], x[0]), reverse=True)
142     else:
143         return sorted(bins.items())
144
145
146 def dot_product(x, y):
147     """Return the sum of the element-wise product of vectors x and y."""
148     return sum(_x * _y for _x, _y in zip(x, y))
149
150
151 def element_wise_product(x, y):
152     """Return vector as an element-wise product of vectors x and y."""
153     assert len(x) == len(y)
154     return np.multiply(x, y)
155
156
157 def matrix_multiplication(x, *y):
158     """Return a matrix as a matrix-multiplication of x and arbitrary number of matrices
    *y."""
```

```
159
160     result = x
161     for _y in y:
162         result = np.matmul(result, _y)
163
164     return result
165
166
167 def vector_add(a, b):
168     """Component-wise addition of two vectors."""
169     return tuple(map(operator.add, a, b))
170
171
172 def scalar_vector_product(x, y):
173     """Return vector as a product of a scalar and a vector"""
174     return np.multiply(x, y)
175
176
177 def probability(p):
178     """Return true with probability p."""
179     return p > random.uniform(0.0, 1.0)
180
181
182 def weighted_sample_with_replacement(n, seq, weights):
183     """Pick n samples from seq at random, with replacement, with the
184     probability of each element in proportion to its corresponding
185     weight."""
186     sample = weighted_sampler(seq, weights)
187     return [sample() for _ in range(n)]
188
189
190 def weighted_sampler(seq, weights):
191     """Return a random-sample function that picks from seq weighted by weights."""
192     totals = []
193     for w in weights:
194         totals.append(w + totals[-1] if totals else w)
195     return lambda: seq[bisect.bisect(totals, random.uniform(0, totals[-1]))]
196
197
198 def weighted_choice(choices):
199     """A weighted version of random.choice"""
200     # NOTE: should be replaced by random.choices if we port to Python 3.6
201
202     total = sum(w for _, w in choices)
203     r = random.uniform(0, total)
204     upto = 0
205     for c, w in choices:
206         if upto + w >= r:
207             return c, w
208         upto += w
209
210
211 def rounder(numbers, d=4):
212     """Round a single number, or sequence of numbers, to d decimal places."""
```

```
213     if isinstance(numbers, (int, float)):
214         return round(numbers, d)
215     else:
216         constructor = type(numbers)  # Can be list, set, tuple, etc.
217         return constructor(rounder(n, d) for n in numbers)
218 
219 
220 def num_or_str(x):  # TODO: rename as `atom`
221     """The argument is a string; convert to a number if possible, or strip it."""
222     try:
223         return int(x)
224     except ValueError:
225         try:
226             return float(x)
227         except ValueError:
228             return str(x).strip()
229 
230 
231 def euclidean_distance(x, y):
232     return np.sqrt(sum((_x - _y) ** 2 for _x, _y in zip(x, y)))
233 
234 
235 def manhattan_distance(x, y):
236     return sum(abs(_x - _y) for _x, _y in zip(x, y))
237 
238 
239 def hamming_distance(x, y):
240     return sum(_x != _y for _x, _y in zip(x, y))
241 
242 
243 def cross_entropy_loss(x, y):
244     return (-1.0 / len(x)) * sum(_x * np.log(_y) + (1 - _x) * np.log(1 - _y) for _x, _y in
    zip(x, y))
245 
246 
247 def mean_squared_error_loss(x, y):
248     return (1.0 / len(x)) * sum((_x - _y) ** 2 for _x, _y in zip(x, y))
249 
250 
251 def rms_error(x, y):
252     return np.sqrt(ms_error(x, y))
253 
254 
255 def ms_error(x, y):
256     return mean((_x - _y) ** 2 for _x, _y in zip(x, y))
257 
258 
259 def mean_error(x, y):
260     return mean(abs(_x - _y) for _x, _y in zip(x, y))
261 
262 
263 def mean_boolean_error(x, y):
264     return mean(_x != _y for _x, _y in zip(x, y))
265 
```

```python

def normalize(dist):
    """Multiply each number by a constant such that the sum is 1.0"""
    if isinstance(dist, dict):
        total = sum(dist.values())
        for key in dist:
            dist[key] = dist[key] / total
            assert 0 <= dist[key] <= 1  # probabilities must be between 0 and 1
        return dist
    total = sum(dist)
    return [(n / total) for n in dist]


def random_weights(min_value, max_value, num_weights):
    return [random.uniform(min_value, max_value) for _ in range(num_weights)]


def sigmoid(x):
    """Return activation value of x with sigmoid function."""
    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(value):
    return value * (1 - value)


def elu(x, alpha=0.01):
    return x if x > 0 else alpha * (np.exp(x) - 1)


def elu_derivative(value, alpha=0.01):
    return 1 if value > 0 else alpha * np.exp(value)


def tanh(x):
    return np.tanh(x)


def tanh_derivative(value):
    return 1 - (value ** 2)


def leaky_relu(x, alpha=0.01):
    return x if x > 0 else alpha * x


def leaky_relu_derivative(value, alpha=0.01):
    return 1 if value > 0 else alpha


def relu(x):
    return max(0, x)


```

```
320 def relu_derivative(value):
321     return 1 if value > 0 else 0
322 
323 
324 def step(x):
325     """Return activation value of x with sign function"""
326     return 1 if x >= 0 else 0
327 
328 
329 def gaussian(mean, st_dev, x):
330     """Given the mean and standard deviation of a distribution, it returns the probability
    of x."""
331     return 1 / (np.sqrt(2 * np.pi) * st_dev) * np.e ** (-0.5 * (float(x - mean) / st_dev) **
    2)
332 
333 
334 def linear_kernel(x, y=None):
335     if y is None:
336         y = x
337     return np.dot(x, y.T)
338 
339 
340 def polynomial_kernel(x, y=None, degree=2.0):
341     if y is None:
342         y = x
343     return (1.0 + np.dot(x, y.T)) ** degree
344 
345 
346 def rbf_kernel(x, y=None, gamma=None):
347     """Radial-basis function kernel (aka squared-exponential kernel)."""
348     if y is None:
349         y = x
350     if gamma is None:
351         gamma = 1.0 / x.shape[1]  # 1.0 / n_features
352     return np.exp(-gamma * (-2.0 * np.dot(x, y.T) +
353                             np.sum(x * x, axis=1).reshape((-1, 1)) + np.sum(y * y,
    axis=1).reshape((1, -1))))
354 
355 
356 # ______________________________________________________________________________
357 # Grid Functions
358 
359 
360 orientations = EAST, NORTH, WEST, SOUTH = [(1, 0), (0, 1), (-1, 0), (0, -1)]
361 turns = LEFT, RIGHT = (+1, -1)
362 
363 
364 def turn_heading(heading, inc, headings=orientations):
365     return headings[(headings.index(heading) + inc) % len(headings)]
366 
367 
368 def turn_right(heading):
369     return turn_heading(heading, RIGHT)
370 
```

```
371
372 def turn_left(heading):
373     return turn_heading(heading, LEFT)
374
375
376 def distance(a, b):
377     """The distance between two (x, y) points."""
378     xA, yA = a
379     xB, yB = b
380     return np.hypot((xA - xB), (yA - yB))
381
382
383 def distance_squared(a, b):
384     """The square of the distance between two (x, y) points."""
385     xA, yA = a
386     xB, yB = b
387     return (xA - xB) ** 2 + (yA - yB) ** 2
388
389
390 # ____________________________________________________________________________
391 # Misc Functions
392
393 class injection:
394     """Dependency injection of temporary values for global functions/classes/etc.
395     E.g., `with injection(DataBase=MockDataBase): ...`"""
396
397     def __init__(self, **kwds):
398         self.new = kwds
399
400     def __enter__(self):
401         self.old = {v: globals()[v] for v in self.new}
402         globals().update(self.new)
403
404     def __exit__(self, type, value, traceback):
405         globals().update(self.old)
406
407
408 def memoize(fn, slot=None, maxsize=32):
409     """Memoize fn: make it remember the computed value for any argument list.
410     If slot is specified, store result in that slot of first argument.
411     If slot is false, use lru_cache for caching the values."""
412     if slot:
413         def memoized_fn(obj, *args):
414             if hasattr(obj, slot):
415                 return getattr(obj, slot)
416             else:
417                 val = fn(obj, *args)
418                 setattr(obj, slot, val)
419                 return val
420     else:
421         @functools.lru_cache(maxsize=maxsize)
422         def memoized_fn(*args):
423             return fn(*args)
424
```

```
425     return memoized_fn
426
427
428 def name(obj):
429     """Try to find some reasonable name for the object."""
430     return (getattr(obj, 'name', 0) or getattr(obj, '__name__', 0) or
431             getattr(getattr(obj, '__class__', 0), '__name__', 0) or
432             str(obj))
433
434
435 def isnumber(x):
436     """Is x a number?"""
437     return hasattr(x, '__int__')
438
439
440 def issequence(x):
441     """Is x a sequence?"""
442     return isinstance(x, collections.abc.Sequence)
443
444
445 def print_table(table, header=None, sep='   ', numfmt='{}'):
446     """Print a list of lists as a table, so that columns line up nicely.
447     header, if specified, will be printed as the first row.
448     numfmt is the format for all numbers; you might want e.g. '{:.2f}'.
449     (If you want different formats in different columns,
450     don't use print_table.) sep is the separator between columns."""
451     justs = ['rjust' if isnumber(x) else 'ljust' for x in table[0]]
452
453     if header:
454         table.insert(0, header)
455
456     table = [[numfmt.format(x) if isnumber(x) else x for x in row]
457              for row in table]
458
459     sizes = list(map(lambda seq: max(map(len, seq)), list(zip(*[map(str, row) for row in
    table]))))
460
461     for row in table:
462         print(sep.join(getattr(str(x), j)(size) for (j, size, x) in zip(justs, sizes, row)))
463
464
465 def open_data(name, mode='r'):
466     aima_root = os.path.dirname(__file__)
467     aima_file = os.path.join(aima_root, *['aima-data', name])
468
469     return open(aima_file, mode=mode)
470
471
472 def failure_test(algorithm, tests):
473     """Grades the given algorithm based on how many tests it passes.
474     Most algorithms have arbitrary output on correct execution, which is difficult
475     to check for correctness. On the other hand, a lot of algorithms output something
476     particular on fail (for example, False, or None).
477     tests is a list with each element in the form: (values, failure_output)."""
```

```
478     return mean(int(algorithm(x) != y) for x, y in tests)
479
480
481 # ____________________________________________________________________________
482 # Expressions
483
484 # See https://docs.python.org/3/reference/expressions.html#operator-precedence
485 # See https://docs.python.org/3/reference/datamodel.html#special-method-names
486
487 class Expr:
488     """A mathematical expression with an operator and 0 or more arguments.
489     op is a str like '+' or 'sin'; args are Expressions.
490     Expr('x') or Symbol('x') creates a symbol (a nullary Expr).
491     Expr('-', x) creates a unary; Expr('+', x, 1) creates a binary."""
492
493     def __init__(self, op, *args):
494         self.op = str(op)
495         self.args = args
496
497     # Operator overloads
498     def __neg__(self):
499         return Expr('-', self)
500
501     def __pos__(self):
502         return Expr('+', self)
503
504     def __invert__(self):
505         return Expr('~', self)
506
507     def __add__(self, rhs):
508         return Expr('+', self, rhs)
509
510     def __sub__(self, rhs):
511         return Expr('-', self, rhs)
512
513     def __mul__(self, rhs):
514         return Expr('*', self, rhs)
515
516     def __pow__(self, rhs):
517         return Expr('**', self, rhs)
518
519     def __mod__(self, rhs):
520         return Expr('%', self, rhs)
521
522     def __and__(self, rhs):
523         return Expr('&', self, rhs)
524
525     def __xor__(self, rhs):
526         return Expr('^', self, rhs)
527
528     def __rshift__(self, rhs):
529         return Expr('>>', self, rhs)
530
531     def __lshift__(self, rhs):
```

```
532         return Expr('<<', self, rhs)
533 
534     def __truediv__(self, rhs):
535         return Expr('/', self, rhs)
536 
537     def __floordiv__(self, rhs):
538         return Expr('//', self, rhs)
539 
540     def __matmul__(self, rhs):
541         return Expr('@', self, rhs)
542 
543     def __or__(self, rhs):
544         """Allow both P | Q, and P |'==>'| Q."""
545         if isinstance(rhs, Expression):
546             return Expr('|', self, rhs)
547         else:
548             return PartialExpr(rhs, self)
549 
550     # Reverse operator overloads
551     def __radd__(self, lhs):
552         return Expr('+', lhs, self)
553 
554     def __rsub__(self, lhs):
555         return Expr('-', lhs, self)
556 
557     def __rmul__(self, lhs):
558         return Expr('*', lhs, self)
559 
560     def __rdiv__(self, lhs):
561         return Expr('/', lhs, self)
562 
563     def __rpow__(self, lhs):
564         return Expr('**', lhs, self)
565 
566     def __rmod__(self, lhs):
567         return Expr('%', lhs, self)
568 
569     def __rand__(self, lhs):
570         return Expr('&', lhs, self)
571 
572     def __rxor__(self, lhs):
573         return Expr('^', lhs, self)
574 
575     def __ror__(self, lhs):
576         return Expr('|', lhs, self)
577 
578     def __rrshift__(self, lhs):
579         return Expr('>>', lhs, self)
580 
581     def __rlshift__(self, lhs):
582         return Expr('<<', lhs, self)
583 
584     def __rtruediv__(self, lhs):
585         return Expr('/', lhs, self)
```

```
586
587     def __rfloordiv__(self, lhs):
588         return Expr('//', lhs, self)
589
590     def __rmatmul__(self, lhs):
591         return Expr('@', lhs, self)
592
593     def __call__(self, *args):
594         """Call: if 'f' is a Symbol, then f(0) == Expr('f', 0)."""
595         if self.args:
596             raise ValueError('Can only do a call for a Symbol, not an Expr')
597         else:
598             return Expr(self.op, *args)
599
600     # Equality and repr
601     def __eq__(self, other):
602         """x == y' evaluates to True or False; does not build an Expr."""
603         return isinstance(other, Expr) and self.op == other.op and self.args == other.args
604
605     def __lt__(self, other):
606         return isinstance(other, Expr) and str(self) < str(other)
607
608     def __hash__(self):
609         return hash(self.op) ^ hash(self.args)
610
611     def __repr__(self):
612         op = self.op
613         args = [str(arg) for arg in self.args]
614         if op.isidentifier():  # f(x) or f(x, y)
615             return '{}({})'.format(op, ', '.join(args)) if args else op
616         elif len(args) == 1:  # -x or -(x + 1)
617             return op + args[0]
618         else:  # (x - y)
619             opp = (' ' + op + ' ')
620             return '(' + opp.join(args) + ')'
621
622
623 # An 'Expression' is either an Expr or a Number.
624 # Symbol is not an explicit type; it is any Expr with 0 args.
625
626
627 Number = (int, float, complex)
628 Expression = (Expr, Number)
629
630
631 def Symbol(name):
632     """A Symbol is just an Expr with no args."""
633     return Expr(name)
634
635
636 def symbols(names):
637     """Return a tuple of Symbols; names is a comma/whitespace delimited str."""
638     return tuple(Symbol(name) for name in names.replace(',', ' ').split())
639
```

```
640
641 def subexpressions(x):
642     """Yield the subexpressions of an Expression (including x itself)."""
643     yield x
644     if isinstance(x, Expr):
645         for arg in x.args:
646             yield from subexpressions(arg)
647
648
649 def arity(expression):
650     """The number of sub-expressions in this expression."""
651     if isinstance(expression, Expr):
652         return len(expression.args)
653     else:  # expression is a number
654         return 0
655
656
657 # For operators that are not defined in Python, we allow new InfixOps:
658
659
660 class PartialExpr:
661     """Given 'P |'==>'| Q, first form PartialExpr('==>', P), then combine with Q."""
662
663     def __init__(self, op, lhs):
664         self.op, self.lhs = op, lhs
665
666     def __or__(self, rhs):
667         return Expr(self.op, self.lhs, rhs)
668
669     def __repr__(self):
670         return "PartialExpr('{}', {})".format(self.op, self.lhs)
671
672
673 def expr(x):
674     """Shortcut to create an Expression. x is a str in which:
675     - identifiers are automatically defined as Symbols.
676     - ==> is treated as an infix |'==>'|, as are <== and <=>.
677     If x is already an Expression, it is returned unchanged. Example:
678     >>> expr('P & Q ==> Q')
679     ((P & Q) ==> Q)
680     """
681     return eval(expr_handle_infix_ops(x), defaultkeydict(Symbol)) if isinstance(x, str) else
    x
682
683
684 infix_ops = '==> <== <=>'.split()
685
686
687 def expr_handle_infix_ops(x):
688     """Given a str, return a new str with ==> replaced by |'==>'|, etc.
689     >>> expr_handle_infix_ops('P ==> Q')
690     "P |'==>'| Q"
691     """
692     for op in infix_ops:
```

```
693         x = x.replace(op, '|' + repr(op) + '|')
694     return x
695 
696 
697 class defaultkeydict(collections.defaultdict):
698     """Like defaultdict, but the default_factory is a function of the key.
699     >>> d = defaultkeydict(len); d['four']
700     4
701     """
702 
703     def __missing__(self, key):
704         self[key] = result = self.default_factory(key)
705         return result
706 
707 
708 class hashabledict(dict):
709     """Allows hashing by representing a dictionary as tuple of key:value pairs.
710     May cause problems as the hash value may change during runtime."""
711 
712     def __hash__(self):
713         return 1
714 
715 
716 # _____________________________________________________________________________
717 # Queues: Stack, FIFOQueue, PriorityQueue
718 # Stack and FIFOQueue are implemented as list and collection.deque
719 # PriorityQueue is implemented here
720 
721 
722 class PriorityQueue:
723     """A Queue in which the minimum (or maximum) element (as determined by f and
724     order) is returned first.
725     If order is 'min', the item with minimum f(x) is
726     returned first; if order is 'max', then it is the item with maximum f(x).
727     Also supports dict-like lookup."""
728 
729     def __init__(self, order='min', f=lambda x: x):
730         self.heap = []
731         if order == 'min':
732             self.f = f
733         elif order == 'max':  # now item with max f(x)
734             self.f = lambda x: -f(x)  # will be popped first
735         else:
736             raise ValueError("Order must be either 'min' or 'max'.")
737 
738     def append(self, item):
739         """Insert item at its correct position."""
740         heapq.heappush(self.heap, (self.f(item), item))
741 
742     def extend(self, items):
743         """Insert each item in items at its correct position."""
744         for item in items:
745             self.append(item)
746 
```

```
747     def pop(self):
748         """Pop and return the item (with min or max f(x) value)
749         depending on the order."""
750         if self.heap:
751             return heapq.heappop(self.heap)[1]
752         else:
753             raise Exception('Trying to pop from empty PriorityQueue.')
754 
755     def __len__(self):
756         """Return current capacity of PriorityQueue."""
757         return len(self.heap)
758 
759     def __contains__(self, key):
760         """Return True if the key is in PriorityQueue."""
761         return any([item == key for _, item in self.heap])
762 
763     def __getitem__(self, key):
764         """Returns the first value associated with key in PriorityQueue.
765         Raises KeyError if key is not present."""
766         for value, item in self.heap:
767             if item == key:
768                 return value
769         raise KeyError(str(key) + " is not in the priority queue")
770 
771     def __delitem__(self, key):
772         """Delete the first occurrence of key."""
773         try:
774             del self.heap[[item == key for _, item in self.heap].index(True)]
775         except ValueError:
776             raise KeyError(str(key) + " is not in the priority queue")
777         heapq.heapify(self.heap)
778 
779 
780 # ____________________________________________________________________________
781 # Useful Shorthands
782 
783 
784 class Bool(int):
785     """Just like `bool`, except values display as 'T' and 'F' instead of 'True' and
    'False'."""
786     __str__ = __repr__ = lambda self: 'T' if self else 'F'
787 
788 
789 T = Bool(True)
790 F = Bool(False)
791 
```