```python
# encoding: utf-8

"""
A Python implementation of the FP-growth algorithm.

Basic usage of the module is very simple:

    > from fp_growth import find_frequent_itemsets
    > find_frequent_itemsets(transactions, minimum_support)
"""

from collections import defaultdict, namedtuple

__author__ = "Eric Naeseth <eric@naeseth.com>"
__copyright__ = "Copyright ÂŠ 2009 Eric Naeseth"
__license__ = "MIT License"


def find_frequent_itemsets(transactions, minimum_support, include_support=False):
    """
    Find frequent itemsets in the given transactions using FP-growth. This
    function returns a generator instead of an eagerly-populated list of items.

    The `transactions` parameter can be any iterable of iterables of items.
    `minimum_support` should be an integer specifying the minimum number of
    occurrences of an itemset for it to be accepted.

    Each item must be hashable (i.e., it must be valid as a member of a
    dictionary or a set).

    If `include_support` is true, yield (itemset, support) pairs instead of
    just the itemsets.
    """
    items = defaultdict(lambda: 0)  # mapping from items to their supports

    # if using support rate instead of support count
    if 0 < minimum_support <= 1:
        minimum_support = minimum_support * len(transactions)

    # Load the passed-in transactions and count the support that individual
    # items have.
    for transaction in transactions:
        for item in transaction:
            items[item] += 1

    # Remove infrequent items from the item support dictionary.
    items = dict(
```

```python
        (item, support) for item, support in items.items() if support >= minimum_support
    )

    # Build our FP-tree. Before any transactions can be added to the tree, they
    # must be stripped of infrequent items and their surviving items must be
    # sorted in decreasing order of frequency.
    def clean_transaction(transaction):
        transaction = filter(lambda v: v in items, transaction)
        transaction = sorted(transaction, key=lambda v: items[v], reverse=True)
        return transaction

    master = FPTree()
    for transaction in list(map(clean_transaction, transactions)):
        master.add(transaction)

    def find_with_suffix(tree, suffix):
        for item, nodes in tree.items():
            support = sum(n.count for n in nodes)
            if support >= minimum_support and item not in suffix:
                # New winner!
                found_set = [item] + suffix
                yield (found_set, support) if include_support else found_set

                # Build a conditional tree and recursively search for frequent
                # itemsets within it.
                cond_tree = conditional_tree_from_paths(tree.prefix_paths(item))
                for s in find_with_suffix(cond_tree, found_set):
                    yield s  # pass along the good news to our caller

    # Search for frequent itemsets, and yield the results we find.
    for itemset in find_with_suffix(master, []):
        yield itemset


class FPTree(object):
    """
    An FP tree.

    This object may only store transaction items that are hashable
    (i.e., all items must be valid as dictionary keys or set members).
    """

    Route = namedtuple("Route", "head tail")

    def __init__(self):
        # The root node of the tree.
```

```
        self._root = FPNode(self, None, None)

        # A dictionary mapping items to the head and tail of a path of
        # "neighbors" that will hit every node containing that item.
        self._routes = {}

    @property
    def root(self):
        """The root node of the tree."""
        return self._root

    def add(self, transaction):
        """Add a transaction to the tree."""
        point = self._root

        for item in transaction:
            next_point = point.search(item)
            if next_point:
                # There is already a node in this tree for the current
                # transaction item; reuse it.
                next_point.increment()
            else:
                # Create a new point and add it as a child of the point we're
                # currently looking at.
                next_point = FPNode(self, item)
                point.add(next_point)

                # Update the route of nodes that contain this item to include
                # our new node.
                self._update_route(next_point)

            point = next_point

    def _update_route(self, point):
        """Add the given node to the route through all nodes for its item."""
        assert self is point.tree

        try:
            route = self._routes[point.item]
            route[1].neighbor = point  # route[1] is the tail
            self._routes[point.item] = self.Route(route[0], point)
        except KeyError:
            # First node for this item; start a new route.
            self._routes[point.item] = self.Route(point, point)

    def items(self):
```

```python
        """
        Generate one 2-tuples for each item represented in the tree. The first
        element of the tuple is the item itself, and the second element is a
        generator that will yield the nodes in the tree that belong to the item.
        """
        for item in self._routes.keys():
            yield (item, self.nodes(item))

    def nodes(self, item):
        """
        Generate the sequence of nodes that contain the given item.
        """

        try:
            node = self._routes[item][0]
        except KeyError:
            return

        while node:
            yield node
            node = node.neighbor

    def prefix_paths(self, item):
        """Generate the prefix paths that end with the given item."""

        def collect_path(node):
            path = []
            while node and not node.root:
                path.append(node)
                node = node.parent
            path.reverse()
            return path

        return (collect_path(node) for node in self.nodes(item))

    def inspect(self):
        print("Tree:")
        self.root.inspect(1)

        print()
        print("Routes:")
        for item, nodes in self.items():
            print("  %r" % item)
            for node in nodes:
                print("    %r" % node)
```

```python
def conditional_tree_from_paths(paths):
    """Build a conditional FP-tree from the given prefix paths."""
    tree = FPTree()
    condition_item = None
    items = set()

    # Import the nodes in the paths into the new tree. Only the counts of the
    # leaf notes matter; the remaining counts will be reconstructed from the
    # leaf counts.
    for path in paths:
        if condition_item is None:
            condition_item = path[-1].item

        point = tree.root
        for node in path:
            next_point = point.search(node.item)
            if not next_point:
                # Add a new node to the tree.
                items.add(node.item)
                count = node.count if node.item == condition_item else 0
                next_point = FPNode(tree, node.item, count)
                point.add(next_point)
                tree._update_route(next_point)
            point = next_point

    assert condition_item is not None

    # Calculate the counts of the non-leaf nodes.
    for path in tree.prefix_paths(condition_item):
        count = path[-1].count
        for node in reversed(path[:-1]):
            node._count += count

    return tree


class FPNode(object):
    """A node in an FP tree."""

    def __init__(self, tree, item, count=1):
        self._tree = tree
        self._item = item
        self._count = count
        self._parent = None
        self._children = {}
```

```python
        self._neighbor = None

    def add(self, child):
        """Add the given FPNode `child` as a child of this node."""

        if not isinstance(child, FPNode):
            raise TypeError("Can only add other FPNodes as children")

        if child.item not in self._children:
            self._children[child.item] = child
            child.parent = self

    def search(self, item):
        """
        Check whether this node contains a child node for the given item.
        If so, that node is returned; otherwise, `None` is returned.
        """
        try:
            return self._children[item]
        except KeyError:
            return None

    def __contains__(self, item):
        return item in self._children

    @property
    def tree(self):
        """The tree in which this node appears."""
        return self._tree

    @property
    def item(self):
        """The item contained in this node."""
        return self._item

    @property
    def count(self):
        """The count associated with this node's item."""
        return self._count

    def increment(self):
        """Increment the count associated with this node's item."""
        if self._count is None:
            raise ValueError("Root nodes have no associated count.")
        self._count += 1
```

```
    @property
    def root(self):
        """True if this node is the root of a tree; false if otherwise."""
        return self._item is None and self._count is None

    @property
    def leaf(self):
        """True if this node is a leaf in the tree; false if otherwise."""
        return len(self._children) == 0

    @property
    def parent(self):
        """The node's parent"""
        return self._parent

    @parent.setter
    def parent(self, value):
        if value is not None and not isinstance(value, FPNode):
            raise TypeError("A node must have an FPNode as a parent.")
        if value and value.tree is not self.tree:
            raise ValueError("Cannot have a parent from another tree.")
        self._parent = value

    @property
    def neighbor(self):
        """
        The node's neighbor; the one with the same value that is "to the right"
        of it in the tree.
        """
        return self._neighbor

    @neighbor.setter
    def neighbor(self, value):
        if value is not None and not isinstance(value, FPNode):
            raise TypeError("A node must have an FPNode as a neighbor.")
        if value and value.tree is not self.tree:
            raise ValueError("Cannot have a neighbor from another tree.")
        self._neighbor = value

    @property
    def children(self):
        """The nodes that are children of this node."""
        return tuple(self._children.values())

    def inspect(self, depth=0):
        print(("  " * depth) + repr(self))
```

```python
        for child in self.children:
            child.inspect(depth + 1)

    def __repr__(self):
        if self.root:
            return "<%s (root)>" % type(self).__name__
        return "<%s %r (%r)>" % (type(self).__name__, self.item, self.count)


def subs(l):
    """
    Used for assoc_rule
    """
    assert type(l) is list
    if len(l) == 1:
        return [l]
    x = subs(l[1:])
    return x + [[l[0]] + y for y in x]


# Association rules
def assoc_rule(freq, min_conf=0.6):
    """
    This assoc_rule must input a dict for itemset -> support rate
    And also can customize your minimum confidence
    """
    assert type(freq) is dict
    result = []
    for item, sup in freq.items():
        for subitem in subs(list(item)):
            sb = [x for x in item if x not in subitem]
            if sb == [] or subitem == []:
                continue
            if len(subitem) == 1 and (subitem[0][0] == "in" or subitem[0][0] == "out"):
                continue
            conf = sup / freq[tuple(subitem)]
            if conf >= min_conf:
                result.append({"from": subitem, "to": sb, "sup": sup, "conf": conf})
    return result


if __name__ == "__main__":
    from optparse import OptionParser
    import csv
```

```python
p = OptionParser(usage="%prog data_file")
p.add_option(
    "-s",
    "--minimum-support",
    dest="minsup",
    type="int",
    help="Minimum itemset support (default: 2)",
)
p.add_option(
    "-n",
    "--numeric",
    dest="numeric",
    action="store_true",
    help="Convert the values in datasets to numerals (default: false)",
)
p.add_option(
    "-c",
    "--minimum-confidence",
    dest="minconf",
    type="float",
    help="Minimum rule confidence (default 0.6)",
)
p.add_option(
    "-f",
    "--find",
    dest="find",
    type="str",
    help="Finding freq(frequency itemsets) or rule(association rules) (default: freq)",
)
p.set_defaults(minsup=2)
p.set_defaults(numeric=False)
p.set_defaults(minconf=0.6)
p.set_defaults(find="freq")
options, args = p.parse_args()

assert options.find == "freq" or options.find == "rule"

if len(args) < 1:
    p.error("must provide the path to a CSV file to read")

transactions = []
with open(args[0]) as database:
    for row in csv.reader(database):
        if options.numeric:
            transaction = []
            for item in row:
                transaction.append(int(item))
```

```python
            transactions.append(transaction)
        else:
            transactions.append(row)

    result = []
    res_for_rul = {}
    for itemset, support in find_frequent_itemsets(transactions, options.minsup, True):
        result.append((itemset, support))
        res_for_rul[tuple(itemset)] = support

    if options.find == "freq":
        result = sorted(result, key=lambda i: i[0])
        for itemset, support in result:
            print(str(itemset) + " " + str(support))
    if options.find == "rule":
        rules = assoc_rule(res_for_rul, options.minconf)
        for ru in rules:
            print(str(ru["from"]) + " -> " + str(ru["to"]))
            print("support = " + str(ru["sup"]) + "confidence = " + str(ru["conf"]))
```