```python
from itertools import chain, combinations,islice
from collections import defaultdict
from time import time
import pandas as pd
import operator


def run_apriori(infile, min_support, min_conf):
    """
    Run the Apriori algorithm. infile is a record iterator.
    Return:
        rtn_items: list of (set, support)
        rtn_rules: list of ((preset, postset), confidence)
    """
    one_cand_set, all_transactions = gen_one_item_cand_set(infile)

    set_count_map = defaultdict(int)  # maintains the count for each set

    one_freq_set, set_count_map = get_items_with_min_support(
        one_cand_set, all_transactions, min_support, set_count_map)

    freq_map, set_count_map = run_apriori_loops(
        one_freq_set, set_count_map, all_transactions, min_support)

    rtn_items = get_frequent_items(set_count_map, freq_map)
    rtn_rules = get_frequent_rules(set_count_map, freq_map, min_conf)

    return rtn_items, rtn_rules


def gen_one_item_cand_set(input_fileator):
    """
    Generate the 1-item candidate sets and a list of all the transactions.
    """
    all_transactions = list()
    one_cand_set = set()
    for record in input_fileator:
        transaction = frozenset(record)
        all_transactions.append(transaction)
        #======================#
        # STRART YOUR CODE HERE  #
        #======================#
        for item in transaction:
            new_set = set()
            new_set.add(item)
            if frozenset(new_set) not in one_cand_set:
                one_cand_set.add(frozenset(new_set))
        #======================#
        #   END YOUR CODE HERE   #
        #======================#
    return one_cand_set, all_transactions


def get_items_with_min_support(item_set, all_transactions, min_support,
                               set_count_map):
    """
    item_set is a set of candidate sets.
    Return a subset of the item_set
    whose elements satisfy the minimum support.
```

```
    """
    rtn = set()
    local_set = defaultdict(int)

    for item in item_set:
        for transaction in all_transactions:
            if item.issubset(transaction):
                set_count_map[item] += 1
                local_set[item] += 1

    #======================#
    # STRART YOUR CODE HERE  #
    #======================#
    for item, count in local_set.items():
        if local_set[item] >= min_support:
            rtn.add(item)
    #======================#
    #   END YOUR CODE HERE   #
    #======================#



    return rtn, set_count_map


def run_apriori_loops(one_cand_set, set_count_map, all_transactions,
                      min_support):
    """
    Return:
        freq_map: a dict
            {<length_of_set_l>: <set_of_frequent_itemsets_of_length_l>}
        set_count_map: updated set_count_map
    """
    freq_map = dict()
    current_l_set = one_cand_set
    i = 1
    #======================#
    # STRART YOUR CODE HERE  #
    #======================#
    while (current_l_set != set([])):
        freq_map[i] = current_l_set
        current_l_set = join_set(current_l_set, i+1)
        current_c_set, set_count_map =
get_items_with_min_support(current_l_set, all_transactions, min_support,
set_count_map)
        current_l_set = current_c_set

        i += 1
    #======================#
    #   END YOUR CODE HERE   #
    #======================#

    return freq_map, set_count_map


def get_frequent_items(set_count_map, freq_map):
    """ Return frequent items as a list. """
    rtn_items = []
    for key, value in freq_map.items():
```

```
119             [(tuple(item), get_support(set_count_map, item))
120              for item in value])
121     return rtn_items
122 
123 
124 def get_frequent_rules(set_count_map, freq_map, min_conf):
125     """ Return frequent rules as a list. """
126     rtn_rules = []
127     for key, value in islice(freq_map.items(),1,None):
128         for item in value:
129             _subsets = map(frozenset, [x for x in subsets(item)])
130             for element in _subsets:
131                 remain = item.difference(element)
132                 if len(remain) > 0:
133                #=======================#
134                # STRART YOUR CODE HERE  #
135                #=======================#
136                     confidence = float(set_count_map[element.union(remain)])
    / float(set_count_map[element])
137                #=======================#
138                #   END YOUR CODE HERE   #
139                #=======================#
140                     if confidence >= min_conf:
141                         rtn_rules.append(
142                             ((tuple(element), tuple(remain)), confidence))
143     return rtn_rules
144 
145 
146 def get_support(set_count_map, item):
147     """ Return the support of an item. """
148     #=======================#
149     # STRART YOUR CODE HERE  #
150     #=======================#
151     sup_item = set_count_map[item]
152     #=======================#
153     #   END YOUR CODE HERE   #
154     #=======================#
155     return sup_item
156 
157 
158 def join_set(s, l):
159     """
160     Join a set with itself .
161     Return a set whose elements are unions of sets in s with length==l.
162     """
163     #=======================#
164     # STRART YOUR CODE HERE  #
165     #=======================#
166     join_set = set()
167     for set_one in s:
168         for set_two in s:
169             if set_one is not set_two:
170                 joint_set = set_one.union(set_two)
171 
172                 if len(joint_set) == l:
173                     join_set.add(joint_set)
174     #=======================#
175     #   END YOUR CODE HERE   #
176     #=======================#
```

```
178
179
180 def subsets(x):
181     """ Return non =-empty subsets of x. """
182     return chain(*[combinations(x, i + 1) for i, a in enumerate(x)])
183
184
185 def print_items_rules(items, rules, ignore_one_item_set=False,
    name_map=None):
186     for item, support in sorted(items, key=operator.itemgetter(1)):
187         if len(item) == 1 and ignore_one_item_set:
188             continue
189         print ('item: ')
190         print (convert_item_to_name(item, name_map), support)
191     print ('\n------------------------ RULES:')
192     for rule, confidence in sorted(
193             rules, key=operator.itemgetter(1)):
194         pre, post = rule
195         print ('Rule: ')
196         print( convert_item_to_name(pre, name_map),
    convert_item_to_name(post, name_map),confidence)
197
198
199 def convert_item_to_name(item, name_map):
200     """ Return the string representation of the item. """
201     if name_map:
202         return ','.join([name_map[x] for x in item])
203     else:
204         return str(item)
205
206
207 def read_data(fname):
208     """ Read from the file and yield a generator. """
209     file_iter = open(fname, 'rU')
210     for line in file_iter:
211         line = line.strip().rstrip(',')
212         record = frozenset(line.split(','))
213         yield record
214
215
216 def read_name_map(fname):
217     """ Read from the file and return a dict mapping ids to names. """
218     df = pd.read_csv(fname, sep=',\t ', header=None, names=['id', 'name'],
219                      engine='python')
220     return df.set_index('id')['name'].to_dict()
221
222
223
```