```
#Evan DePosit
#New Beginnings
#capstone
#this file contains class defintions for Reading Group Schedule, Staff Schedule, 
Teacher, Events and the maximum mathing biparatite graph algorithm

#
----------------------------------------------------------------------------------------------------
#                                                       $match
#
----------------------------------------------------------------------------------------------------

class Graph():
    def __init__(self):
        self.queue=[]
        self.U=None
        self.V=None
        self.E=None
        self.unionVU=[]

        #for vertex in U:
            #self.unionVU.insert(vertex.num, vertex)

        #for vertex in V:
            #self.unionVU.insert(vertex.num, vertex)


    def print_matches(self):
        print('vetex->mate')
        for vertex in self.unionVU:
            if (vertex.mate):
                print('{} {} {} {} {}'.format('v', vertex.num, '->', vertex.mate.num, 
'   '))
            else:
                print('{} {} {} {} {}'.format('v', vertex.num, '->', 'N', '   '))
        print('')

    def print_queue(self):
        print('queue: ', end='')
        for v in self.queue:
            v.print_vertex()
        print('')

    def print_set(self, part):
        print('set ', end='')
        for v in part:
            v.print_vertex()
        print('')

    def init_queue(self):
        self.queue=[]
        for vertex in self.V:
            if vertex.mate is None:
                self.queue.append(vertex)

    def remove_labels(self):
        for vertex in self.V:
            vertex.label=None
        for vertex in self.U:
            vertex.label=None

    def print_debug(self, loc):
        print('{} {}'.format(loc, 'vetex->mate'))
        for vertex in self.unionVU:
            if (vertex.mate):
                print('{} {} {} {}'.format(vertex.num, '->', vertex.mate.num, '  '))
            else:
                print('{} {} {} {}'.format(vertex.num, '->', 'N', '  '))
        print('')

    def max_match(self):
    # Maximum Matching in Bipartite Graph Algorithm
    # the purpose of this function is to match up teachers with reading groups.
    # another function will generate edges based on times the teacher is available
    #and which reading levels they can work with
        self.init_queue()
        while self.queue:
            w= self.queue.pop(0)
            #change to my own search function if time
            if w in self.V:
                for u in self.E[w.num]:
                    #if u is free in list of vertices connected to w
                    if u.mate is None:
                        w.mate=u
                        u.mate=w
                        #following labeling, umatching, etc will take place after 
finding last free
                        v=w
                        while(v.label is not None):
                            u=v.label
                            if((u.mate == v) and (v.mate== u)):
                                v.mate=None
                                u.mate=None
                            v=u.label
                            v.mate=u
                            u.mate=v
                        self.remove_labels()
                        self.init_queue()
                        #break from for loop because at end of traversal
                        break
                    else:
                        if((w.mate != u) and (u.mate != w) and (u.label is None)):
                            u.label= w
                            self.queue.append(u)
            #else: w in U and matched
            else:
                #label the mate v of w with "w"
                w.mate.label= w
                #enqueue(Q, v) v as in mate v of w?
                self.queue.append(w.mate)
        return


#
----------------------------------------------------------------------------------------------------
#                                                           $act $event $vertex
#
----------------------------------------------------------------------------------------------------

class Vertex():
#vertex will be parent class of reading activities and scheduled events classes
    def __init__(self):
        self.num=None
        self.label=None
        self.mate=None

    def print_vertex(self):
        print(self.num, end=' ')



#classes
class Activity(Vertex):
    def __init__(self, actType):
        super().__init__()
        self.type=actType
        eventTime=None

class Reading_Group_Activity(Activity):
    def __init__(self, group, day, actType):
        super().__init__(actType)
        self.readingGroup= group
        self.readingLevel=None
        self.day=day

       # self.groupNumber= groupNumber
       # self.studentList=[]
       # self.activityList=[]

    def print_act(self):
        print('Group: ', self.readingGroup.groupNumber)
        print('Day: ', self.day)
        print('Activity Type: ', self.type)
        #if self.num:
        #    print('vertex number: ', self.num)
        print('vertex number: ', self.num)

class Event(Vertex):
    def __init__(self, day, start, end, eventType, teacher=None):
        super().__init__()
        self.day= day
        self.start=start
        self.end= end
        self.teacher=teacher
        #can be either pointer to one student or group of students
        self.type=eventType
        self.students=None

    def print_event(self):
        print()
        print('Event: ', self.type)
        if self.teacher:
            print('Teacher: ', self.teacher.name)
        if self.day != None:
            print('day: ', self.day)
        if self.start and self.end:
            print('Start: ', tm.min_to_time(self.start), 'End: ',
tm.min_to_time(self.end))
        if self.num:
            print('vertex number: ', self.num)


#
----------------------------------------------------------------------------------------------------
#                                                           $sched
#
```

```
----------------------------------------------------------------------------------------------------

class Reading_Group_Sched(Graph):
    def __init__(self, teacherSchedule, classList, scheduleTimes):
        super().__init__()
        self.teacherSchedule=teacherSchedule
        self.classList=classList
        #schedule times is schedule_parameters object
        self.schedParams=scheduleTimes
        #dictionary key= day, item list of events on that day
        self.eventSched={}
        #dictionary key= day, item list of activities of all groups
        self.actSched={}

    def print_group_teacher(self):
        weekLen= self.schedParams.days

        for groupNumber in self.classList.groupNumberList:
            self.classList.readingGroups[groupNumber]
            print('Group', self.classList.readingGroups[groupNumber].groupNumber,
'Activity Event Match')
            for act in self.classList.readingGroups[groupNumber].activityList:
                act.print_act()
                print()
                if act.mate:
                    act.mate.print_event()
                else:
                    print('NO MATCH')
                print()
                print()
            print()
            print()
            print()

    def make_group_event(self):
        """
        input:self used to access teacher objects
        output: creates event, adds it to list for teachers and master events list
        """
        weekLen= self.schedParams.days
        allEvents=[]

        #interate through teachers
        for teacher in self.teacherSchedule.teacherList:
            #print(teacher.name)

            #iterate through each day in teachers schedule
            for day in range(0, weekLen):

                teacherDayEventList=[]

                #get event times for that day from parameters object
                #get in/out times for that day from teacher class
                eventTimes= self.schedParams.get_days_eTime(day)
                teacherTimes= teacher.get_days_inOuts(day)

                # test print e times and teacher times to see if they line up
                #self.schedParams.print_days_eTimes(day)
                #print('day ', day, 'teacher availability')
                #print(teacherTimes)

                #interate over list of inout times` and comapre to event times for
```

```
each day
                for event in eventTimes:
                    for inOut in teacherTimes:
                        #if list is not empty
                        #compare if in time is less than or equal to event start
                        #and out time is greater than or equal to event end time
                        if inOut and inOut[0] <= event[0] and inOut[1]>= event[1]:
                            #create event make_event(day, start, stop, teacher)
                            newEvent= Event(day, event[0], event[1], "Small Group
Lesson", teacher)

                            teacherDayEventList.append(newEvent)

                            #add to master list doesn't have to be function can just
append it to
                            allEvents.append(newEvent)
                            #newEvent.print_event()

                #add to teachers list
                teacher.add_event(teacherDayEventList, day)

        #allEvents not organized by day need separate function
        #self.groupEventList= allEvents
        self.add_events(allEvents)
        random.shuffle(allEvents)
        self.V= allEvents


    def add_events(self, eventList):
    #input: list of events
    #output: dictionary of events by day
    #function interates through list of events makes list for each day, list is not
organized, just for max match
        for event in eventList:
            if event.day in self.eventSched:
                self.eventSched[event.day].append(event)
            else:
                self.eventSched[event.day]= []
                self.eventSched[event.day].append(event)

    #generate activity list for each readingGroup
    def make_group_act(self):
        weekLen= self.schedParams.days
        classActList=[]

        for groupNumber in self.classList.groupNumberList:
            groupActList=[]

            for day in range(0, weekLen):
                #reading_group_Act(self, group, day, actType):
                newAct=
Reading_Group_Activity(self.classList.readingGroups[groupNumber], day, 'Small Group
Lesson')
                #newAct.print_act()
                #add each act to groups list and sched act list for maxMatch
                groupActList.append(newAct)
                classActList.append(newAct)

            #add group act list to group class object. in group function add it to
each stuent too
            self.classList.readingGroups[groupNumber].add_actList(groupActList)
        #add group act list to class schedule (dictionary day:actList) if keeping as
a list instead of dictionary by day
        self.add_act_list(classActList)

        #add group act to set u in graph class
        random.shuffle(classActList)
        self.U= classActList

        #print('U in make grup act funct')
        #self.print_set(self.U)
        #self.print_act_list(self.U)

    def print_act_list(self, actList):
        print('print act list')
        for act in actList:
            act.print_act()


    def add_act_list(self, classActList):
    #input: unordered list of all activities created for every group
    #output: add activities to Reading Group sched.actSched by day
        for activity in classActList:
            if activity.day in self.actSched:
                self.actSched[activity.day].append(activity)
            else:
                self.actSched[activity.day]=[]
                self.actSched[activity.day].append(activity)

    def add_teacher_pref_test(self):
        str1= 'For each staff member, enter the group number of each group that may
be scheduled with the staff member'
        str2= 'separating each group number with a spae and entering return when
finished'
        str3= 'If all groups may be scheduled with the staff member, enter all'

        GroupsNumbers=[]

        print('{} {} {}'.format(str1, str2, str3))

        for teacher in self.teacherSchedule.teacherList:

            #remove break when finished testing
            break

            groupPref=[]

            line=input(teacher.name +': ')
            if 'all' in line or 'All' in line:
                #add group pref to teacher
                allGroups= self.classList.numOfGroups
                for i in range(1, allGroups+1):
                    groupPref.append(i)
            else:
                numList=line.split(' ')
                for numStr in numList:
                    isNumber= re.match('^\d+$', numStr)
                    if isNumber:
                        num= int(numStr)
                        groupPref.append(num)
                #add group pef to teacher
            teacher.groupPref=groupPref
            #print(groupPref)
        #hard code teacher pref for repeated testing
        self.teacherSchedule.teacherList[0].groupPref= [1,2,3]
        self.teacherSchedule.teacherList[1].groupPref= [1,2,3]
        self.teacherSchedule.teacherList[2].groupPref= [3]

    def add_teacher_pref(self):
        str1= 'For each staff member, enter the group number of each group that may
be scheduled with the staff member'
        str2= 'separating each group number with a spae and entering return when
finished'
        str3= 'If all groups may be scheduled with the staff member, enter all'

        GroupsNumbers=[]

        print('{} {} {}'.format(str1, str2, str3))

        for teacher in self.teacherSchedule.teacherList:

            #remove break when finished testing

            groupPref=[]

            line=input(teacher.name +': ')
            if 'all' in line or 'All' in line:
                #add group pref to teacher
                allGroups= self.classList.numOfGroups
                for i in range(1, allGroups+1):
                    groupPref.append(i)
            else:
                numList=line.split(' ')
                for numStr in numList:
                    isNumber= re.match('^\d+$', numStr)
                    if isNumber:
                        num= int(numStr)
                        groupPref.append(num)
                #add group pef to teacher
            teacher.groupPref=groupPref
            #print(groupPref)

        #for teacher in self.teacherSchedule.teacherList:
            #print(teacher.name, teacher.groupPref)


    def set_edges(self):
        vertexCount=0
        weekLen=self.schedParams.days
        edgeList={}

        #number vertexes
        for i in range(0, weekLen):
            for event in self.eventSched[i]:
                event.num=vertexCount
                vertexCount= vertexCount+1

        for i in range(0, weekLen):
            for act in self.actSched[i]:
                act.num=vertexCount
                vertexCount= vertexCount+1

        for teacher in self.teacherSchedule.teacherList:
            pref= teacher.groupPref
            for day in range(0, weekLen):
                for event in teacher.lessonEventSched[day]:
                    for act in self.actSched[day]:
                        if act.readingGroup.groupNumber in pref:
                            if act.num in edgeList:
                                edgeList[act.num].append(event)
                            else:
                                edgeList[act.num]=[]
                                edgeList[act.num].append(event)
                            if event.num in edgeList:
                                edgeList[event.num].append(act)
                            else:
                                edgeList[event.num]=[]
                                edgeList[event.num].append(act)

                            #print('match')
                            #act.print_act()
                            #event.print_event()
                            #print()
            self.E=edgeList


def input_sched_testTimes(numberOfDays):

    #set set weeks eTimes
    weekTimes=[]

    for i in range(0, numberOfDays):
        start1=tm.time_to_min('11:15')
        end1=tm.time_to_min('11:55')
        start2=tm.time_to_min('12:40')
        end2=tm.time_to_min('1:20')
        dayTimes=[]
        dayTimes.append(start1)
        dayTimes.append(end1)
        dayTimes.append(start2)
        dayTimes.append(end2)
        weekTimes.append(dayTimes)

    #test with extra padding
    #test with less parameters
    #start1=time_to_min('11:15')
    #end1=time_to_min('11:55')
    #dayTimes=[]
    #dayTimes.append(start1)
    #dayTimes.append(end1)
    #weekTimes.append(dayTimes)
    return weekTimes

def input_sched_Times(numberOfDays):
    #set set weeks eTimes
    str1 ='For each day, enter a start time, followed by a stop time for to indicate 
when the morning and afternoon sessions'
    str2= 'or the reading time should begin and end. If there is no afternoon start 
and end time, press enter.'
    print(str1 + '\n' + str2)

    weekTimes=[]

    for i in range(0, numberOfDays):
        print('day:', i)
        start1= tm.time_to_min(input('Morning start time:'))
        end1= tm.time_to_min(input('Morning end time:'))

        start2= tm.time_to_min(input('Afternoon start time:'))
        end2= tm.time_to_min(input('Afternoon end time:'))

        dayTimes=[]
        if start1 and end1:
            dayTimes.append(start1)
            dayTimes.append(end1)
        if start2 and end2:
            dayTimes.append(start2)
            dayTimes.append(end2)

        weekTimes.append(dayTimes)

    #test with extra padding
    #test with less parameters
    #start1=time_to_min('11:15')
    #end1=time_to_min('11:55')
    #dayTimes=[]
    #dayTimes.append(start1)
    #dayTimes.append(end1)
    #weekTimes.append(dayTimes)
    return weekTimes

class Schedule_Parameters():
    #def __init__(self, days, actPerDay, duration, start1, end1, start2=None,
end2=None):
    def __init__(self, days, actPerDay, duration):
        self.days=days
        self.actPerDay=actPerDay
        self.duration = duration
        #list of of days, each day is  list of start and stop times
        self.dailyEvents=[]

    def week_len(self):
        return self.days

    def set_weeks_eTimes(self, startEndList):
        #input: list of start and end times for each day of the week
        #output: list of event times for each day
        for times in startEndList:
            start1 = times[0]
            end1 = times[1]
            if len(times) > 2:
                start2= times[2]
                end2= times[3]
            else:
               start2=None
               end2 =None
            daysEvents=[]
            daysEvents= self.set_days_eTimes(start1, end1, start2, end2)
            self.dailyEvents.append(daysEvents)

    def set_days_eTimes(self, start1, end1, start2, end2):
        daysEvents=[]
        event=0
        periodStart=start1
        periodEnd=end1
        actStart=None
        actEnd=None
        nonScheduled=self.actPerDay
        #figure out how many activities in first time chunk

        actInPeriod=int((periodEnd-periodStart)/self.duration)
        while(actInPeriod and nonScheduled):
            actStart=periodStart
            actEnd= periodStart+self.duration
            #add atart and end times to event times list
            startEnd=[]
            startEnd.append(actStart)
            startEnd.append(actEnd)

            daysEvents.append(startEnd)

            #update variables for next time through loop
            nonScheduled=nonScheduled-1
            #print('start: ', actStart)
            #print('end: ', actEnd)
            #print(startEnd)
            #print('remaining events to plan: ', nonScheduled)
            #add to list of event times
            periodStart= actEnd
            actInPeriod=int((periodEnd-periodStart)/self.duration)

        if(nonScheduled and start2):
            periodStart=start2
            periodEnd=end2
            actStart=None
            actEnd=None

            actInPeriod=int((periodEnd-periodStart)/self.duration)
            while(actInPeriod and nonScheduled):
                actStart=periodStart
                actEnd= periodStart+self.duration
                #add atart and end times to event times list
                startEnd=[]
                startEnd.append(actStart)
                startEnd.append(actEnd)
                daysEvents.append(startEnd)
                #update variables for next time through loop
                nonScheduled=nonScheduled-1
                #print('start: ', actStart)
                #print('end: ', actEnd)
                #print(startEnd)
                #print('remaining events to plan: ', nonScheduled)
                #add to list of event times
                periodStart= actEnd
                actInPeriod=int((periodEnd-periodStart)/self.duration)

        #for event in self.eventTimes:
            #print(event)
        return daysEvents

    def print_days_eTimes(self, day):
        """
        input: day that exist in list of events times for each day in schedule
        output: void, prints list of events on that day
        """
        print('day', day, 'event times')
        if(self.dailyEvents[day]):
            print(self.dailyEvents[day])
        else:
            print('list empty')
    def print_all_eTimes(self):
        print('All events times for week')
        print(self.dailyEvents)

    def get_days_eTime(self, day):
        """
        input day
        output list of n elements, each element is list of start and stop time of
that  event
        """
        return self.dailyEvents[day]

#---------------------------------------------------------------------------------------------------
#                                                         $teacher
#
----------------------------------------------------------------------------------------------------

class Staff_Schedule():
    def __init__(self):
        self.dayCount=0
        self.maxTimesInOut=0
        self.teacherList=[]

    def read_teachers(self, filePath):
        #throw away first line or maybe use to determine day
        fin = open(filePath, 'rt')
        line=fin.readline()
        #line=line[:-1:]
        #print(line)

        teacherData=[]

        teacherDataList=[]
        while True:
            #read in each student by line and count total
            line= fin.readline()
            if not line:
                break
            #line=line[:-1:]
            #print each line of student data
            #print(line)

            #add to teacherData list
            teacherData=line.split(',')
            teacherDataList.append(teacherData)

        fin.close()
        #print(teacherDataList)
        return teacherDataList

    #add pattern matching to identify how many in/out times for eac day
    def teacher_sched(self, teacherTimes):
        #input: list of strings each string lis line from teacher schedule file
        #output list of teacher objects
        #start by just making general schedule for one day/all week
        teacherList=[]
        #print('teacherTimes list')
        #print(teacherTimes)
        #orgainize teacher schedule by teacher or day?  teacher
        #count how many days in schedule and how many in/out times in day from file
        #need to have class set up before this point
        #hard coded, need to add functions
        self.dayCount=4
        self.maxTimesInOut=2
        dayCount=self.dayCount
        inOutCount=self.maxTimesInOut

        for line in teacherTimes:
            #list of days, each day is list of in/out lists
            dayList=[]
            name =line.pop(0)
            #list of days, each day is list of time chunks which is list  clockin/out
times

            #loop for each day
            for i in range(0, dayCount):
                #each day is list of in/out times
                dayList.append([])
                #get each time that teacher enters/leaves class in one day
                for j in range(0, inOutCount):
                    #get in and out Time as string
                    inTimeStr=line.pop(0)
                    outTimeStr=line.pop(0)
                    #convert int time to min int times with function
                    #need handling for empty strngs if:else intime=NONE
                    inOut=[]
                    if(inTimeStr):
                        inTime=tm.time_to_min(inTimeStr)
                        outTime=tm.time_to_min(outTimeStr)
                        #add to list tuple list
                        inOut.append(inTime)
                        inOut.append(outTime)
                        #add to tuple to lis day list
                    dayList[i].append(inOut)

            #change to separate function
            #print(name)
            #print each day
            #for i in range(0, dayCount):
                #print('day', i)
                #for j in range(0, inOutCount):
                #for j in range(0, len(dayList[i])):
                    #if(dayList[i][j]):
                        #print('In: ', dayList[i][j][0])
                        #print('Out: ', dayList[i][j][1])

            self.teacherList.append(Teacher(name, dayList, dayCount))
        return

    def print_staff(self):
        print('print staff function')
        for teacher in self.teacherList:
            teacher.print_teacher()

    def sched_to_file(self, weekLen):

        fout=open('teacher_sched.csv', 'wt')
        fout.close()
        for teacher in self.teacherList:
            teacherLines=teacher.sched_to_file(weekLen)

            temp=dict(teacherLines[0])
            headers=list(temp.keys())
            #print(headers)

            fout=open('teacher_sched.csv', 'at')
            fout.write(teacher.name + '\n')
            cout = csv.DictWriter(fout, headers)
            cout.writeheader()
            cout.writerows(teacherLines)
            fout.close()

        #with open('student_sched.csv', 'wt') as fout:
            #cout = csv.DictWriter(fout, headers)
            #cout.writeheader()
            #cout.writerows(studentLines)

class Teacher():
    def __init__(self, name, schedule, weekLen):
        self.name=name
        self.schedule=schedule
        #dictionary is schedule with day as key and list of lesson events as item
        self.lessonEventSched={}
        self.groupPref=[]

    def print_teacher(self):
        print(self.name)
        day=0
        for i in self.schedule:
            print('day ', day)
            inOut= 0
            for time in self.schedule[day]:
                if self.schedule[day][inOut]:
                    print(self.schedule[day][inOut][0])
                    print(self.schedule[day][inOut][1])
                inOut= inOut + 1
            day=day + 1

    def get_days_inOuts(self, day):
        return self.schedule[day]

    def add_event(self, eventList, day):
        self.lessonEventSched[day]=eventList


    def sched_to_file(self, weekLen):
        fileLines=[]
        startTimeList=[]

        # lessonEventSched

        #go through all schedules to get lst of all the times
        for day in range(0, weekLen):
            for event in self.lessonEventSched[day]:
                if event.start not in startTimeList:
                    startTimeList.append(event.start)
        startTimeList.sort()

        #print(self.name)
        #print(startTimeList)

        #initalize dictionary just to avoide using if else statements later
        schedTime={}
        for time in startTimeList:
            schedTime[time]={}
            schedTime[time]['Time']=tm.min_to_time(time)

        #go through student sched dictionary again
        for day in range(0, weekLen):
            for event in self.lessonEventSched[day]:
                if event.mate:
                    schedTime[event.start]['Day '+ str(event.day+1)]=event.type + '-
Group' + str(event.mate.readingGroup.groupNumber)
                else:
                    schedTime[event.start]['Day '+ str(event.day+1)]= 'No Group 
Scheduled'
        #fileLines.append(self.fullName)
        for time in startTimeList:
            #print(schedTime[time])
            fileLines.append(schedTime[time])
        #print(fileLines)
        return fileLines

import timeConvert as tm
import student as st
import csv
import random
import re
```