CEE 505

Jean-Luc Jackson

11/16/2016

Midterm Project

The Graph Problem – Database Version

Problem Statement

Part 1: Design a database representation of the "Graph Problem"

- *Design an SQL database to represent a flight network.*
- *Document your design with a database layout graph.*
- *Develop and document a program that reads the provided input and generates a "Graph.db" sqlite3 database.*

Part 2: Adapt the Path Search Algorithm to work off the database rather than private variables.

- *Develop a tool that connects to your "Graph.db" through a Graph class.*
- *Do not store lines or nodes in your Python class, but rather get that information using SELECT statements.*
- *Replace functions like getAttachedLines() or getAttachedNodes() with SELECT statements.*

Part 1: Database Creation

Code Description – FlightDB

Solution Overview

A database named '*Graph.db*' was created using a Python code, namely by creating a *FlightDB* object. This *FlightDB* class reads the two provided comma-separated value files (*AirportData.csv* and *FlightData.csv*) in a function named *FileToDB* and adds each row to tables with names corresponding to the CSV filename. Therefore, upon initiation of a *FlightDB* object, the 'Graph.db' database contains two tables.

Database Layout

The '*Graph.db*' database was designed to appear as a flight itinerary while providing an efficient storage capability, namely having one table to store all routes and each route's individual flights. The columns in the table are: *row id*, *From*, *Via*, *To*, *RouteID*, *Seq*, and *Trip Time*. Columns *From* and *To* are airport codes; *Via* is a flight number from the *From* airport to the *To* airport; *RouteID* is an integer identifier for each route; *Seq* is an integer that identifies the sequence of flights for a given route; and *Trip Time* is the time duration for each flight step or overall route, depending on the row, stored as an integer. Example rows are shown in Table 1 below.

The table groups rows by route number. Each group begins with a *Summary Row* which displays the origin and destination cities, the *RouteID*, and the total *Trip Time* for that route. Columns *Via* and *Seq* are left intentionally blank for display purposes, but this fact is also utilized later in SQL SELECT statements. After the initial *Summary Row*, all subsequent rows are *Flight Step Rows* which match the description in the paragraph above, with every column containing information.

One table is created for each overall trip from the origin city to the destination city. This table has the general name format of *Origin Airport Code* + to + *Destination Airport Code*. For example, if findPaths

were called for Seattle to Miami and again for San Francisco to Denver, there would be two tables in the database, *SEAtoMIA* and *SFOtoDIA*.

Table 1. 'Graph.db' Database Layout

	TYPE:	*INTEGER*	CHAR(3)	TEXT	CHAR(3)	INTEGER	INTEGER	INTEGER
Row Type		**id**	**From**	**Via**	**To**	**RouteID**	**Seq**	**Trip Time**
Summary Row		*1*	*ORIGIN*		*DEST*	*1*		*TOTAL*
Flight Step Row		*2*	*ORIGIN*	*Flight1*	*City2*	*1*	*1*	*Trip Time*
Flight Step Row		*3*	*City2*	*Flight2*	*City3*	*1*	*2*	*Trip Time*
⋮		⋮	⋮	⋮	⋮	⋮	⋮	⋮
Summary Row		*9*	*ORIGIN*		*DEST*	*3*		*TOTAL*
Flight Step Row		*10*	*ORIGIN*	*Flight1*	*City2*	*3*	*1*	*Trip Time*
Flight Step Row		*11*	*City2*	*Flight2*	*City3*	*3*	*2*	*Trip Time*
⋮		⋮	⋮	⋮	⋮	⋮	⋮	⋮
Flight Step Row		*15*	*City6*	*Flight6*	*DEST*	*3*	*6*	*Trip Time*

Implementation Details

File Reading

Much like previous homework programs, this program opens a file and reads it line by line. Each line is processed and used to create a row in a database table. The string methods *split* and *strip* are used to separate the comma-separated values and remove new-line characters.

SQL Code

Appendix 2: SQL Code for Part 1 shows the SQL code used to create the database described above. The *FlightDB* class utilizes TABLE functions CREATE, INSERT INTO, and DROP IF. SQL string statements are stored in functions and returned for use by cursor *execute()* functions. Prepared statements are also utilized, placing question marks (*?*) into the SQL string to be replaced by elements in a list which are also passed in the *execute()* function. All attempts to create or insert data into a database are done in a try-except format in the attempt to catch any IOErrors or database errors that may occur.

Part 2: Database-Enabled Path Search Algorithm

Code Description – Midterm_Graph

Solution Overview

This *Midterm_Graph* class connects to the database created in Part 1 and uses this database's information to replace all local variable storage. There is a *Path* class that acts as a history storage center: both line and node histories are stored in lists for each *Path* object as well as a length integer. This class also handles mathematical operations such as addition and subtraction so that *Paths* can be combined or reduced. Aside from this *Path* class, all other information is accessed through SELECT statements to the '*Graph.db*' database. The function *findPaths()* does the majority of the work in this class as it contains the path search algorithm. It also uses SQL string statements for information accessing and table creation and updating.

Path Search Algorithm

The *Midterm_Graph* class can calculate all possible paths between two requested nodes. This code is a modified version of the code from HW#4 to be database-compatible. Upon calling *findPaths*, possible paths are found using recursion taking an initial node and destination node as arguments. The recursive process occurs as follows:

1) Step on the provided node and get information about that node.
2) Find all connecting lines and travel down these lines.
3) The next node can fall into 3 categories:
    a) Nodes visited before
        i) Do nothing. Return to previous node.
    b) The destination node
        i) Save the route it took to arrive here.
    c) An unvisited intermediate node
        i) Find all lines attached to this node.
        ii) Create a new instance of the *findPaths()* recursion beginning at the nodes at the other ends of these lines.

This function creates a *Path* object that documents the path traveled along this recursive process. *Path* objects are created and added up along the travel of *findPaths()*, documenting the node and line histories. These possible paths are stored in a list called *allPaths*. At the end of the recursion when the list contains all possible paths between the two nodes, this list of paths is inserted into the database table documenting that trip's routes using *savePathsToDB()*.

Subsequent methods *findShortestPath()* and *findLongestPath()* utilize the database storing all routes. As described in Part 1, each pair of origin-destination nodes is used as a string key that is mapped to a database table containing all possible paths between this node pair. These shortest/longest path methods use this database table and SELECT statements to find the shortest *Trip Time* value existing in a *Summary Row*. The RouteID pertaining to the shortest route is passed to *printRoute()* so that the proper route is printed, as described in the next section.

Route Printing

Multiple SQL statements and string concatenation are utilized to compile descriptive printed lines for a flight itinerary to the console. The length is processed from an integer value, as stored in the database,

to a string in standard HR:MN format. Since only airport codes are stored in the flight itinerary table, SELECT statements are used to retrieve corresponding cities and states for airports along the route. Output format was based on Professor Mackenzie's examples online for ease in verification

Exception Handling

As described in Part 1, all attempts to create a database cursor or to create or insert data into a database are done in a try-except format.

Code Verification

The following output were generated for comparison with the online verification output.

Shortest Path: SFO to LGA

```
Trip: SFO | San Francisco, CA  to  LGA | New York, NY
Total length of time traveling: 20:15 hours
Flight #1: AK3248 from San Francisco, CA (SFO) departing at 19:45, arriving in
Seattle, WA (SEA) at 21:55
Flight #2: AK1256 from Seattle, WA (SEA) departing at 7:15, arriving in Chicago, IL
(ORD) at 10:55
Flight #3: AA345 from Chicago, IL (ORD) departing at 11:25, arriving in New York, NY
(JFK) at 13:55
Flight #4: CBE from New York, NY (JFK) departing at 15:00, arriving in New York, NY
(LGA) at 16:00
```

Shortest Path: LGA to SFO

```
Trip: LGA | New York, NY  to  SFO | San Francisco, CA
Total length of time traveling: 6:45 hours
Flight #1: UA345 from New York, NY (LGA) departing at 11:30, arriving in Chicago, IL
(ORD) at 14:05
Flight #2: UA49 from Chicago, IL (ORD) departing at 14:50, arriving in San Francisco,
CA (SFO) at 18:15
```

Longest Path: SEA to MIA

```
Trip: SEA | Seattle, WA  to  MIA | Miami, FL
Total length of time traveling: 128:20 hours
Flight #1: SWA10 from Seattle, WA (SEA) departing at 8:45, arriving in Oakland, CA
(OAK) at 11:15
Flight #2: DL882 from Oakland, CA (OAK) departing at 11:55, arriving in Dallas, TX
(DFW) at 15:10
Flight #3: DL1214 from Dallas, TX (DFW) departing at 12:40, arriving in Salt Lake
City, UT (SLC) at 14:05
Flight #4: DL382 from Salt Lake City, UT (SLC) departing at 12:35, arriving in
Chicago, IL (ORD) at 14:55
Flight #5: AA345 from Chicago, IL (ORD) departing at 11:25, arriving in New York, NY
(JFK) at 13:55
Flight #6: CBA from New York, NY (JFK) departing at 7:00, arriving in New York, NY
(LGA) at 8:00
Flight #7: AA734 from New York, NY (LGA) departing at 14:55, arriving in Denver, CO
(DIA) at 17:45
Flight #8: SWA125 from Denver, CO (DIA) departing at 13:25, arriving in Miami, FL
(MIA) at 17:05
```

Longest Path: MIA to SEA

```
Trip: MIA | Miami, FL  to  SEA | Seattle, WA
Total length of time traveling: 106:35 hours
Flight #1: UA768 from Miami, FL (MIA) departing at 8:05, arriving in Dallas, TX (DFW)
at 10:49
Flight #2: DL1214 from Dallas, TX (DFW) departing at 12:40, arriving in Salt Lake
City, UT (SLC) at 14:05
Flight #3: DL382 from Salt Lake City, UT (SLC) departing at 12:35, arriving in
Chicago, IL (ORD) at 14:55
Flight #4: AA345 from Chicago, IL (ORD) departing at 11:25, arriving in New York, NY
(JFK) at 13:55
Flight #5: CBA from New York, NY (JFK) departing at 7:00, arriving in New York, NY
(LGA) at 8:00
Flight #6: AA734 from New York, NY (LGA) departing at 14:55, arriving in Denver, CO
(DIA) at 17:45
Flight #7: AK246 from Denver, CO (DIA) departing at 16:10, arriving in Seattle, WA
(SEA) at 18:40
```

Appendices

Appendix 1: Python Code for Part 1

FlightDB Class

```
import sys
import os
import sqlite3 as DBI


class FlightDB(object):

    def __init__(self,folder_name):

        os.chdir( folder_name )
        self.folder_name = folder_name
        self.dbFilename = 'Graph.db'

        # Delete database from previous run first
        try:
            os.remove(self.dbFilename)
            print "Last run's database deleted."
        except OSError:
            pass

        # Connect to database at provided address
        try:
            self.db = DBI.connect(self.dbFilename)
            self.cu = self.db.cursor()
            self.db.text_factory = str
            print "Connected to database at {}.".format(self.dbFilename)
        except DBI.Error as e:
            print "sqlite3 failed with error: {}".format(e)
            sys.exit(1)

        # Create Table from FlightData
        try:
            print "flights"
            self.cu.executescript(self.createFlightTable())
            self.db.commit()
            print "FlightData table created."
        except DBI.Error as e:
            print "sqlite3 failed with error: {}".format(e)
            sys.exit(1)

        # Create Table from AirportData
        try:
            print "airports"
            self.cu.executescript(self.createAirportTable())
            self.db.commit()
            print "AirportData table created."
        except DBI.Error as e:
            print "sqlite3 failed with error: {}".format(e)
            sys.exit(1)
```

```
        self.FileToDB(self.folder_name)
        self.closeDB()


    def FileToDB(self,folder_name):
        # Read .csv file and INSERT each line into AirportData table
        try:
            fIn = open('AirportData.csv','r')
            firstLine = fIn.readline().split(',')
            headers = [item.strip() for item in firstLine]
            indeces = range(1,len(headers)+1)
            mapper = dict(zip(headers,indeces))
            mapper[0] = 'id'
            print "Headers = ", mapper

            for line in fIn:
                splitted = line.split(',')
                stripped = [l.strip('\n') for l in splitted]
                print "Line = ", stripped
                self.cu.execute(self.addToAirportTable(),stripped)
                self.db.commit()

            print "File ({}) used to create database.\n".format(folder_name)
        except IOError:
            print "Could not open file for reading at: ({})".format(folder_name)
            sys.exit(1)

        try:
            fIn = open('FlightData.csv','r')
            firstLine = fIn.readline().split(',')
            headers = [item.strip() for item in firstLine]
            indeces = range(1,len(headers)+1)
            mapper = dict(zip(headers,indeces))
            mapper[0] = 'id'
            print "Headers = ", mapper

            for line in fIn:
                splitted = line.split(',')
                stripped = [l.strip('\n') for l in splitted]
                print "Line = ", stripped
                self.cu.execute(self.addToFlightTable(),stripped)
                self.db.commit()

            print "File ({}) used to create database.\n".format(folder_name)
        except IOError:
            print "Could not open file for reading at: ({})".format(folder_name)
            sys.exit(1)


    def closeDB(self):
        self.db.close()
```

Appendix 2: SQL Code for Part 1

Referenced by FlightDB Class

```
    def addToAirportTable(self):
        return """
            INSERT INTO AirportData ( Airport, City, State, longitude, latitude, x, y
)
            VALUES ( ? , ? , ? , ? , ? , ? , ? );
            """

    def addToFlightTable(self):
        return """
            INSERT INTO FlightData ( 'flight number', Operator, [From], [To], Depart,
Arrival )
            VALUES ( ? , ? , ? , ? , ? , ? );
            """

    def createAirportTable(self):
        return """
            DROP TABLE IF EXISTS AirportData;

            CREATE TABLE AirportData (
                id            INTEGER not null primary key AUTOINCREMENT,
                Airport       TEXT,
                City          CHAR(25),
                State         TEXT,
                longitude     FLOAT(4),
                latitude      FLOAT(4),
                x             FLOAT(3),
                y             FLOAT(3)
                );
            """

    def createFlightTable(self):
        return """
            DROP TABLE IF EXISTS FlightData;

            CREATE TABLE FlightData (
                id              INTEGER not null primary key AUTOINCREMENT,
                'flight number' TEXT,
                Operator        CHAR(25),
                [From]          TEXT,
                [To]            TEXT,
                Depart          TIME,
                Arrival         TIME
                );
            """
```

Appendix 3: Python & SQL Code for Part 2

Midterm_Graph Class

SQL Code included throughout:

```
#Import Directories
import os
import sys
import sqlite3 as DBI
import copy

# Import Path Classes
from Path import *


class Midterm_Graph(object):
    '''
    Class Hosts A Graph Object.
    Reads Data From Provided .txt File
    Creates/Stores A Series of Node-Line Relationships
    Contained Attributes - Line Objects, Node Objects
    Contained Algorithms - Path Finder, Shortest Path Finder
    '''


    def __init__(self, folder_name):
        '''
        Creates a Graph object by reading from 'Graph.db' database located in
folder_name.
        '''
        # Changes Directory To Folder Location
        os.chdir( folder_name )
        self.folder_name = folder_name
        self.dbFilename = 'Graph.db'


        # Nodes and Lines already exist in database at DBAddress.
        # Connect to database at provided address
        try:
            self.db = DBI.connect(self.dbFilename)
            self.cu = self.db.cursor()
            self.db.text_factory = str
            print "Connected to database at {}.".format(self.dbFilename)
        except DBI.Error as e:
            print "sqlite3 failed with error: {}".format(e)
            sys.exit(1)


    def __str__(self):
        pass


    def findPaths(self, startNode, destNode, pastPath=Path(),
allPaths=[],first=True):
        '''
```

```
        This Method Travels Down Lines Until destNode Has Been Reached.
        Returns List Containing Path Objects Which Reached destNodeID.
        Takes arguments: -startNode, From Which Subsequent Paths Are Found
                         -destNode, Final Node We're Seeking As Our Destination
                         -pastPath, Path We've Traveled Up To This Point
                         -allPaths, List Containing Path Objects Seeking destNote
        '''
        # If it's the first recursion (just got called), clear the previous findPath
data
        if first:
            allPaths = []
            # Create database table with this Trip's name
            self.createTripTable('{}to{}'.format(startNode,destNode))


        # If We've Visited This Node Before, Don't Go Any Further
        # Return An Empty List
        for nodeName in pastPath.getNodeHistory():
            if startNode == nodeName:
                #print "Line has already been stepped through."
                lastLine = pastPath.getLineHistory()[-1]
                lastLineLength = self.DBgetLineLength(lastLine)

                # remove last line and length so we don't save it
                pastPath -= Path([],[lastLine],lastLineLength)
                return []

        # If We've Reached Our Destination, Don't Go Any Further
        # Return The Path Up to This Point
        if startNode == destNode:
            # Add Destination Node to Path (Step On Node)
            pastPath += Path([startNode],[],0.0)
            # Find The Last Line And Node
            lastLine = pastPath.getLineHistory()[-1]
            lastLineLength = self.DBgetLineLength(lastLine)
            lastNode = pastPath.getNodeHistory()[-1]

            # Save finalPath
            finalPath = copy.deepcopy(pastPath)

            #Remove Previous Line, Node, And Line Length
            pastPath -= Path([lastNode],[lastLine],lastLineLength)

            return [finalPath]

        # If This Node Has Not Been Visited
        # Add Current Node to Path (Step On Node)
        pastPath += Path([startNode],[],0.0)
        self.cu.execute("""
            SELECT f.'flight number'
            FROM FlightData as f
            WHERE f.[From] = ?
            """, [startNode])
        lineIDs = [row[0] for row in self.cu]
```

```
        # Find Path For Each Line Attached to startNode
        for line in lineIDs:
            # Get node (airport) we're traveling to from this line (flight) we're on
            self.cu.execute("""
                SELECT f.[To] FROM FlightData as f
                WHERE f.'flight number' = ?
                """,[line])
            endNode = self.cu.fetchone()[0]

            # find this line's length
            lineLength = self.DBgetLineLength(line)
            # Calculate layover by difference in last flight's arrival and this
flight's depart
            # Takes into account that if a layover is < 30 mins we add 24hrs
            try: # if this isn't the first flight, there will be a line history
                lastFlight = pastPath.getLineHistory()[-1]
                layover = self.DBgetLayoverTime(lastFlight,line)
            except IndexError:
                # if it's the first flight, there will be no history, so no layover
                layover = 0

            lineLength += layover

            tempPath = pastPath + Path([],[line],lineLength)
            nextPath = self.findPaths(endNode,destNode,tempPath,allPaths,False)

            allPathsIDs = [p.getID() for p in allPaths]
            if len(nextPath) == 1 and nextPath[0] not in allPaths:
                allPaths.append(nextPath[0])

        if first:
            self.savePathsToDB(allPaths)
        return allPaths

    def savePathsToDB(self,allPaths):
        # Build a string that represents the Trip name
        # Each trip gets its own table in database
        try:
            initialNode = allPaths[0].getNodeHistory()[0]
            lastNode = allPaths[0].getNodeHistory()[-1]
            tripName = "{}to{}".format(initialNode,lastNode)
        except IndexError:
            return

        # allPaths is a list containing Path objects.
        # Loop through this list of Paths and add them to the database
        RouteID = 0
        for path in allPaths:
            RouteID += 1
            nodeHistory = path.getNodeHistory()
            lineHistory = path.getLineHistory()
            totalLength = path.getLength()

            # Insert trip summary at first row of each Route group
            tripSummary = [ initialNode, lastNode,  RouteID ]
```

```
            self.cu.execute("""
                INSERT INTO {} ([From] , [To] , RouteID)
                VALUES ( ? , ? , ? )
                """.format(tripName),tripSummary)

            # Loop through the lineHistory of this current Path and add
            # each flight in the history to the proper table in the database.
            seq = 0
            for line in lineHistory:
                fromNode = nodeHistory[seq]
                toNode = nodeHistory[seq+1]
                via = lineHistory[seq]
                stepLength = self.DBgetLineLength(line)
                toSave = [fromNode, via, toNode, RouteID, seq + 1, stepLength]
                seeql = """
                    INSERT INTO
                    {}     ([From] , [Via] , [To] , RouteID , Seq , 'Trip Time')
                    VALUES ( ?      , ?     , ?    , ?      , ?   , ? )
                    """.format(tripName)
                self.cu.execute(seeql,toSave)
                seq += 1

            self.cu.execute("""
                UPDATE {} SET 'Trip Time' = {}
                WHERE RouteID = ?
                AND Via is NULL;
                """.format(tripName,totalLength),[RouteID])

        self.db.commit()
        print "\nAll possible routes from {} to {} saved to
database.\n".format(initialNode,lastNode)


    def findShortestPath(self, startID, endID):
        '''
        Find smallest length value from Trip Summary rows
        SELECT the row with the smallest 'Trip Time' where there is no Via value
        Convert time integer back to time time
        '''
        try:
            db = DBI.connect(self.dbFilename)
            cu = self.db.cursor()
        except DBI.Error as e:
            print "sqlite3 failed with error: {}".format(e)
            sys.exit(1)

        tripName = '{}to{}'.format(startID,endID)

        # Find row with minimum length
        cu.execute("""
            SELECT t.RouteID
            FROM {} as t
            WHERE t.'Trip Time' in
                (SELECT MIN(t.'Trip Time') FROM {} as t
                 WHERE t.Via is NULL);
```

```
        """.format(tripName,tripName))
    RouteID = cu.fetchone()[0]

    return self.printRoute(startID,endID,RouteID)


def findLongestPath(self, startID, endID):
    '''
    Find smallest length value from Trip Summary rows
    SELECT the row with the smallest 'Trip Time' where there is no Via value
    Convert time integer back to time time
    '''
    try:
        db = DBI.connect(self.dbFilename)
        cu = self.db.cursor()
    except DBI.Error as e:
        print "sqlite3 failed with error: {}".format(e)
        sys.exit(1)

    tripName = '{}to{}'.format(startID,endID)

    # Find row with minimum length
    cu.execute("""
        SELECT t.RouteID
        FROM {} as t
        WHERE t.'Trip Time' in
            (SELECT MAX(t.'Trip Time') FROM {} as t
             WHERE t.Via is NULL);
        """.format(tripName,tripName))
    RouteID = cu.fetchone()[0]

    return self.printRoute(startID,endID,RouteID)


def printRoute(self,startID,endID,RouteID):
    # Prints the path of a given RouteID
    try:
        db = DBI.connect(self.dbFilename)
        cu = self.db.cursor()
    except DBI.Error as e:
        print "sqlite3 failed with error: {}".format(e)
        sys.exit(1)

    tripName = '{}to{}'.format(startID,endID)

    # Find Summary Row of interest
    cu.execute("""
        SELECT t.'Trip Time'
        FROM {} as t
        WHERE t.RouteID = ?
        AND t.Via is NULL;
        """.format(tripName),[RouteID])
    length = cu.fetchone()[0]

    hours = (length/60)
```

```
mins = (length%60)
timeString = str(hours) + ':' + str(mins)

# Get INITIAL city&state
cu.execute("""
    SELECT a.City, a.State
    FROM AirportData as a
    WHERE a.Airport = ?;
    """,[startID])
[startCity, startSta] = [i for i in cu.fetchone()]
# Get DEST city&state
cu.execute("""
    SELECT a.City, a.State
    FROM AirportData as a
    WHERE a.Airport = ?;
    """,[endID])
[endCity, endSta] = [i for i in cu.fetchone()]

s = '\n'
# Get whole trip itinerary for shortest route
cu.execute("""
    SELECT t.Seq, t.Via, t.[From], t.[To], t.'Trip Time'
    FROM {} as t
    WHERE t.RouteID = ?
    """.format(tripName),[RouteID])
for row in cu.fetchall():
    if None in row: # Route Summary row contains 'None's
        s += 'Trip: {} | {}, {}  to  {} | {}, {}\n'.\
        format(startID,startCity,startSta,endID,endCity,endSta)
        s += 'Total length of time traveling: {} hours\n'.format(timeString)
    else: # Grab and label data for printing
        seq = row[0]
        via = row[1]
        fro = row[2]
        to = row[3]
        time = row[4]
        cu.execute("""
            SELECT a.City, a.State, f.Depart, f.Arrival
            FROM AirportData as a, FlightData as f
            WHERE a.Airport = ?
            AND f.'flight number' = ?;
            """,[fro,via])
        [fromCity, fromSta,depart,arrive] = [i for i in cu.fetchone()]
        # Get TO city&state
        cu.execute("""
            SELECT a.City, a.State
            FROM AirportData as a
            WHERE a.Airport = ?;
            """,[to])
        [toCity, toSta] = [i for i in cu.fetchone()]

        s += 'Flight #{}: {} from {}, {} ({}) departing at {}, '\
        .format(seq,via,fromCity,fromSta,fro,depart)
        s += 'arriving in {}, {} ({}) at {}\n'.format(toCity,toSta,to,arrive)
```

```
    return s


def createTripTable(self,name):
    self.cu.executescript("""
        DROP TABLE IF EXISTS {};
        CREATE TABLE {} (
            id            INTEGER not null primary key AUTOINCREMENT,
            [From]        CHAR(3),
            Via           TEXT,
            [To]          CHAR(3),
            RouteID       INTEGER,
            Seq           INTEGER,
            'Trip Time'   INTEGER
        );
        """.format(name,name))


def DBgetLineLength(self, line):
    try:
        db = DBI.connect(self.dbFilename)
        cu = self.db.cursor()
    except DBI.Error as e:
        print "sqlite3 failed with error: {}".format(e)
        sys.exit(1)

    cu.execute("""
        SELECT f.Depart, f.Arrival
        FROM FlightData as f
        WHERE f.'flight number' = ?;
        """,[line])

    [departAt, arriveAt] = [i for i in cu.fetchone()]
    depSplit = departAt.split(':')
    depInt = int(depSplit[0])*60 + int(depSplit[1])
    arrSplit = arriveAt.split(':')
    arrInt = int(arrSplit[0])*60 + int(arrSplit[1])

    return arrInt - depInt

def DBgetLayoverTime(self,lastFlight,nextFlight):
    try:
        db = DBI.connect(self.dbFilename)
        cu = db.cursor()
    except DBI.Error as e:
        print "sqlite3 failed with error: {}".format(e)
        sys.exit(1)

    # get time integer of last flight's arrival
    cu.execute("""
        SELECT f.Arrival
        FROM FlightData as f
        WHERE f.'flight number' = ?
        """,[lastFlight])
    lastArrival = cu.fetchone()[0]
```

```
arrSplit = lastArrival.split(':')
arrInt = int(arrSplit[0])*60 + int(arrSplit[1])

# get time integer of next flight's departure
cu.execute("""
    SELECT f.Depart
    FROM FlightData as f
    WHERE f.'flight number' = ?
    """,[nextFlight])
nextDepart = cu.fetchone()[0]
depSplit = nextDepart.split(':')
depInt = int(depSplit[0])*60 + int(depSplit[1])

# calculate layover
layover = depInt - arrInt
if (layover < 30) or (layover < 0): # if we missed our flight
    layover += 24*60                # our trip just got 24 hrs longer

return layover
```

Path Class

```
class Path(object):
    '''
    Class Hosts a Path Object.
    Contained Attributes - Path ID, Length, and Node/Line Histories.
    '''

    def __init__(self, nodeHistory=[], lineHistory=[], length=0):

        # Initialize List of Nodes Traveled - Contains NodeIDs (strings)
        self.nodeHistory = nodeHistory
        # Initialize List of Lines Traveled - Contains LineIDs (strings)
        self.lineHistory = lineHistory
        # Initialize length of time to zero (time integer value)
        self.length = int(length)

        # Create pathID
        s = ''
        for i in range(0,len(nodeHistory)):
            s += nodeHistory[i] + ' -> '
        self.ID = '( ' + s[:len(s) - 4] + ' )'

    '''----------------------------------------
    ACTION CALLS
    use informational calls
    to perform calculations
    and return requested value
    ----------------------------------------'''
    def __str__(self): # Print Statement
        s = "Path {} has a length {} and has seen {} nodes and {} lines."\
        .format(self.ID,self.length, len(self.nodeHistory), len(self.lineHistory))
        return s

    def __add__(self,otherPath): # Add Error If Trying To Add Integer/Float
        # Step Through Line - Add Line to lineHistory
        newLineHistory = self.lineHistory + otherPath.getLineHistory()
        # Step On Node - Add Node to nodeHistory
        newNodeHistory = self.nodeHistory + otherPath.getNodeHistory()
        # Increase Path's Length
        newLength = self.length + otherPath.getLength()
        return Path(newNodeHistory, newLineHistory, newLength)

    def __sub__(self,otherPath): # Add Error If Trying To Subtract Integer/Float
        # Step Back Through Line - Remove Line From lineHistory
        newLineHistory = self.lineHistory
        newNodeHistory = self.nodeHistory
        for line in otherPath.getLineHistory():
            newLineHistory = self.lineHistory
            self.lineHistory.remove(line)
        # Step On Node - Removed Node From nodeHistory
        for node in otherPath.getNodeHistory():
            newNodeHistory = self.nodeHistory
            self.nodeHistory.remove(node)
        # Decrease Path's Length
        newLength = self.length - otherPath.getLength()
```

```
        return Path(newNodeHistory, newLineHistory, newLength)

    '''----------------------------------------
    INFORMATIONAL CALLS
    Path histories are stored locally until they can be saved into the database.
    These calls return locally-stored information.
    ----------------------------------------'''

    def getID(self):
        return self.ID

    def getLength(self):
        # Return the Current Length
        # LENGTH IS TIME IN THIS PROGRAM
        return self.length

    def getNodeHistory(self):
        # Return Current nodeHistory
        return self.nodeHistory

    def getLineHistory(self):
        # Return Current lineHistory
        return self.lineHistory
```