Documentation for Programming Project

```python
1   import openpyxl
2   import itertools
3
4   fileName = input('Name of file you want to input: ')
5   wb = openpyxl.load_workbook(fileName)
6
7   wb = openpyxl.load_workbook('TestingData (1NF-5NF).xlsx')
8   wb = openpyxl.load_workbook('TestingData (5NF violation).xlsx')
9
10  sheet = wb.active
11
12  tables = []
13
14  # Parses from xlsx file by checking for borders to indicate a cell of a table and following the border to trace the table.
15
16  tableCount = -1
17  prevCoord = -1
18  tables.append([])
19  rowCount = 0
20  for cell in sheet['A']:
21      if cell.border.top.style or cell.border.left.style or cell.border.right.style or cell.border.bottom.style:
22          row = cell.coordinate[1:]
23          if cell.coordinate[1:] != '1':
24              if int(prevCoord) != int(row) - 1:
25                  tableCount += 1
26                  tables.append([])
27                  tables[tableCount].append([])
28                  rowCount = 0
29              else:
30                  tables[tableCount].append([])
31                  rowCount += 1
32          for nest1 in sheet[row]:
33              if nest1.value != None:
34                  tables[tableCount][rowCount].append(nest1.value)
35          prevCoord = cell.coordinate[1:]
36
37  normalFormChoice = str(input('Normal form level to normalize to (integer or \'bcnf\' for bcnf): '))
```

The purpose of the first 37 lines of code is to take input from the given .xlsx file and parse it. It uses the openpyxl library to identify the cells in the .xlsx file and distinguish unbordered cells with bordered cells. It follows the bordered cells to identify tables and store them in a list (to form a 3 dimensional list). It also takes user input from the command line to determine which normal form the program should converting the tables to.

First Normal Form:

```python
39  # 1nf ------------------------------------------------
40
41  if normalFormChoice == '1' or normalFormChoice == '2' or normalFormChoice == '3' or normalFormChoice == '4' or normalFormChoice == '5' or normalFormChoice == "bcnf":
42      # Inputs primary keys
43      tableChoice = int(input('which table to normalize?'))
44      primaryKey = input('Primary Key (type \'done\' when finished): ')
45      primaryKeys = [primaryKey]
46      while primaryKey != 'done':
47          primaryKey = input('Primary Key (type \'done\' when finished): ')
48          if primaryKey != 'done':
49              primaryKeys.append(primaryKey)
50
51      originalPKs = primaryKeys.copy()
52
53      primaryKeyx = []
54      for value in primaryKeys:
55          primaryKeyx.append(tables[tableChoice][0].index(value))
```

For the 1nf section of the program, lines 42 to 53 take user input to decide which table in the .xlsx file needs to be normalized. It then takes the attributes for the primary key fo the selected table as input and finds the column number of the primary keys.

```
57      # Identifies multivalued coords by looking for curly braces.
58      multivaluedxCoords = []
59      for i in range(len(tables[tableChoice])):
60          for j in range(len(tables[tableChoice][i])):
61              if '{' in str(tables[tableChoice][i][j]) and '}' in str(tables[tableChoice][i][j]):
62                  if j not in multivaluedxCoords:
63                      multivaluedxCoords.append(j)
64      multivaluedxCoords.sort(key=lambda x: x)
```

Lines 58 to 64 looks for multivalued attributes by looking for curly braces ({}) in each cell of the table. It then stores the column numbers of these multivalued attributes.

```
66      # Creates new tables for 1nf and populates them.
67      firstNFTables = []
68      firstNFTableCount = -1
69      for i in multivaluedxCoords:
70          row = 1
71          pullRow = 1
72          firstNFTableCount += 1
73          firstNFTables.append([])
74          firstNFTables[firstNFTableCount].append([])
75          firstNFTables[firstNFTableCount][0].extend(primaryKeys)
76          firstNFTables[firstNFTableCount][0].append(tables[tableChoice][0][i])
77          for j in range(1, len(tables[tableChoice])):
78              if '{' in str(tables[tableChoice][j][i]) and '}' in str(tables[tableChoice][j][i]):
79                  multivalues = tables[tableChoice][j][i][tables[tableChoice][j][i].index('{')+1:tables[tableChoice][j][i].index('}')].split(', ')
80                  for k in multivalues:
81                      firstNFTables[firstNFTableCount].append([])
82                      for l in primaryKeyx:
83                          firstNFTables[firstNFTableCount][row].append(tables[tableChoice][pullRow][l])
84                      firstNFTables[firstNFTableCount][row].append(k)
85                      row += 1
86              else:
87                  firstNFTables[firstNFTableCount].append([])
88                  for l in primaryKeyx:
89                      firstNFTables[firstNFTableCount][row].append(tables[tableChoice][pullRow][l])
90                  firstNFTables[firstNFTableCount][row].append(tables[tableChoice][pullRow][i])
91                  row += 1
92              pullRow += 1
```

Lines 67 to 92 creates a new 3 dimensional list to populate the result of first normalization. It creates a new table for each multivalued attribute and populates it with the primary key and multivalues in such a way that every attribute in each table is now atomic.

```
    multivaluedAtts = []
    for i in multivaluedxCoords:
        if i not in multivaluedAtts:
            multivaluedAtts.append(i)

    # Populates original table once all multivalued attributes are removed.
    firstNFTables.append([])
    firstNFTableCount += 1
    for i in range(len(tables[tableChoice])):
        firstNFTables[firstNFTableCount].append([])
        for j in range(len(tables[tableChoice][i])):
            if j not in multivaluedAtts:
                firstNFTables[firstNFTableCount][i].append(tables[tableChoice][i][j])

    primaryKeysAfter1nf = []

    for i in range(len(firstNFTables) - 1):
        primaryKeysAfter1nf.append(firstNFTables[i][0])

    primaryKeysAfter1nf.append(primaryKeys)

    print('----------')
    print('1nf:')
    for i in firstNFTables:
        print('----------')
        for j in i:
            print(j)
        print('Primary keys:', primaryKeysAfter1nf[firstNFTables.index(i)])
```

The rest of 1nf populates the remaining attributes that were already atomic and prints the result of 1nf out in an organized way.

Second Normal Form:

```
# 2nf --------------------------------------------------

if normalFormChoice == '2' or normalFormChoice == '3' or normalFormChoice == '4' or normalFormChoice == '5' or normalFormChoice == "bcnf":
    secondNFTables = firstNFTables.copy()
    primaryKeysAfter2nf = primaryKeysAfter1nf.copy()
    appendCount = len(secondNFTables)

    # Identifies and parses functinoal dependencies from input.
    FDLeftList = []
    FDRightList = []
    FD = ''
    while FD != 'done':
        FD = input("Functional Dependency (\'done\' if done): ")
        if FD == 'done':
            break
        splitFD = FD.split(' -> ')
        if '{' in splitFD[0] and '}' in splitFD[0]:
            FDLeftList.append(splitFD[0][splitFD[0].index('{')+1:splitFD[0].index('}')].split(', '))
        else:
            FDLeftList.append(splitFD[0].split(', '))
        if '{' in splitFD[1] and '}' in splitFD[1]:
            FDRightList.append(splitFD[1][splitFD[1].index('{')+1:splitFD[1].index('}')].split(', '))
        else:
            FDRightList.append(splitFD[1].split(', '))
```

For the 2nf section, lines 126 to 147 deal with the creation of a new 3 dimensional list for 2nf and inputting of functional dependencies. The code breaks the functional dependencies into two lists, the left side and right side of the dependencies.

```
149     for i in range(len(firstNFTables)):
150         for j in range(len(FDLeftList)):
151             # First checks if all attributes are in table, then checks if left matches primary keys, then checks if any attribute exists in primary key.
152             isIn = True
153             for k in FDLeftList[j]:
154                 if k not in firstNFTables[i][0]:
155                     isIn = False
156                     break
157             if isIn == False:
158                 continue
159             for k in FDRightList[j]:
160                 if k not in firstNFTables[i][0]:
161                     isIn = False
162                     break
163             if isIn == False:
164                 continue
165             if primaryKeysAfter1nf[i] == FDLeftList[j]:
166                 continue
167             isIn = False
168             for k in FDLeftList[j]:
169                 if k in primaryKeysAfter1nf[i]:
170                     isIn = True
171                     break
172             if isIn == False:
173                 continue
174             secondNFTables.append([])
175             primaryKeysAfter2nf.append([])
176             for k in range(len(firstNFTables[i])):
177                 secondNFTables[appendCount].append([])
178             for k in FDLeftList[j]:
179                 indexToAdd = firstNFTables[i][0].index(k)
180                 primaryKeysAfter2nf[appendCount].append(k)
181                 for l in range(len(secondNFTables[appendCount])):
182                     secondNFTables[appendCount][l].append(secondNFTables[i][l][indexToAdd])
183             for k in FDRightList[j]:
184                 indexToAdd = firstNFTables[i][0].index(k)
185                 for l in range(len(secondNFTables[appendCount])):
186                     pop2nf = secondNFTables[i][l].pop(indexToAdd)
187                     secondNFTables[appendCount][l].append(pop2nf)
188             # Checks if there are redundant tuples after moving to new table. Removes redundant tuples.
189             removeRedundancyFromAppend = []
190             [removeRedundancyFromAppend.append(x) for x in secondNFTables[appendCount] if x not in removeRedundancyFromAppend]
191             secondNFTables.pop(appendCount)
192             secondNFTables.append(removeRedundancyFromAppend)
193             if secondNFTables[i][0] == primaryKeysAfter2nf[i]:
194                 secondNFTables.pop(i)
195                 primaryKeysAfter2nf.pop(i)
196             appendCount += 1
```

The main part of 2nf deals with breaking the 1nf tables down and populating the new 2nf list. It loops through the functional dependencies and tables to check if all the attributes in the dependency are present in a given table and checks if the left side of the dependencies only form part of the primary key of a table. It then creates new tables with the attributes that are participating in the partial functional dependency and removes the non-prime attributes from the original table. It also checks for and removes redundancies from the original table.

```
198     print('----------')
199     print('2nf:')
200     for i in secondNFTables:
201         print('----------')
202         for j in i:
203             print(j)
204         print('Primary keys:', primaryKeysAfter2nf[secondNFTables.index(i)])
```

It then prints the results of the 2nf normalization in an orderly fashion.

Third Normal Form:

```python
# 3nf ---------------------------------------------------

# Check if all attributes are in table, then check if left has no primary keys, then loop through other FDs, check if all their valu
# j (outer) makes sure left values aren't in primary key. k (inner) makes sure left values are in pk

if normalFormChoice == '3':
    thirdNFTables = secondNFTables.copy()
    primaryKeysAfter3nf = primaryKeysAfter2nf.copy()
    appendCount = len(thirdNFTables)

    # Performs normalization on any functional dependencies that don't have left side as superkey or right side as prime attribute.
    for i in thirdNFTables:
        for j in range(len(FDLeftList)):
            isIn = True
            for k in FDLeftList[j]:
                if k not in thirdNFTables[thirdNFTables.index(i)][0]:
                    isIn = False
                    break
            if isIn == False:
                continue
            for k in FDRightList[j]:
                if k not in thirdNFTables[thirdNFTables.index(i)][0]:
                    isIn = False
                    break
            if isIn == False:
                continue
            if primaryKeysAfter3nf[thirdNFTables.index(i)] == FDLeftList[j]:
                continue
            isIn = False
            for k in FDRightList[j]:
                if k in primaryKeysAfter3nf[thirdNFTables.index(i)]:
                    isIn = True
                    break
            if isIn == True:
                continue
            for k in range(len(FDLeftList)):
                if k == j:
                    continue
                isIn = True
                for l in FDLeftList[k]:
                    if l not in thirdNFTables[thirdNFTables.index(i)][0]:
                        isIn = False
                        break
                if isIn == False:
                    continue
                for l in FDRightList[k]:
                    if l not in thirdNFTables[thirdNFTables.index(i)][0]:
                        isIn = False
                        break
                if isIn == False:
                    continue
                if primaryKeysAfter3nf[thirdNFTables.index(i)] != FDLeftList[k]:
                    continue
                isIn = True
                for l in FDLeftList[j]:
                    if l not in FDRightList[k]:
                        isIn = False
                        break
                if isIn == False:
                    continue
                thirdNFTables.append([])
                primaryKeysAfter3nf.append([])
                for l in range(len(thirdNFTables[thirdNFTables.index(i)])):
                    thirdNFTables[appendCount].append([])
                for l in FDLeftList[j]:
                    indexToAdd = thirdNFTables[thirdNFTables.index(i)][0].index(l)
                    primaryKeysAfter3nf[appendCount].append(l)
                    for m in range(len(thirdNFTables[appendCount])):
                        thirdNFTables[appendCount][m].append(thirdNFTables[thirdNFTables.index(i)][m][indexToAdd])
                for l in FDRightList[j]:
                    indexToAdd = thirdNFTables[thirdNFTables.index(i)][0].index(l)
                    for m in range(len(thirdNFTables[appendCount])):
                        pop3nf = secondNFTables[thirdNFTables.index(i)][m].pop(indexToAdd)
                        thirdNFTables[appendCount][m].append(pop3nf)
                appendCount += 1
```

Lines 217 to 280, which are the main part of the 3nf code, ensure that the left side of each functional dependency forms a superkey of the table or the right side is prime attributes of the table. This way, it removes any transitive functional dependencies from the table. If there are any transitive functional dependencies, it splits the table to create a new one for the transitive dependency and removes the right side attributes of the violating dependency from the original relation.

```
    print('----------')
    print('3nf:')
    for i in thirdNFTables:
        print('----------')
        for j in i:
            print(j)
        print('Primary keys:', primaryKeysAfter3nf[thirdNFTables.index(i)])
    print(len(primaryKeysAfter3nf))
```

It then prints the results of 3nf in an orderly fashion.

Boyce Codd Normal Form:

```python
# bcnf -----------------------------------------------------

if normalFormChoice == '4' or normalFormChoice == '5' or normalFormChoice == "bcnf":
    thirdNFTables = secondNFTables.copy()
    primaryKeysAfter3nf = primaryKeysAfter2nf.copy()
    appendCount = len(thirdNFTables)

    # Performs normalization on any functional dependencies that don't have left side as superkey.
    for i in thirdNFTables:
        for j in range(len(FDLeftList)):
            isIn = True
            for k in FDLeftList[j]:
                if k not in thirdNFTables[thirdNFTables.index(i)][0]:
                    isIn = False
                    break
            if isIn == False:
                continue
            for k in FDRightList[j]:
                if k not in thirdNFTables[thirdNFTables.index(i)][0]:
                    isIn = False
                    break
            if isIn == False:
                continue
            if primaryKeysAfter3nf[thirdNFTables.index(i)] == FDLeftList[j]:
                continue
            for k in range(len(FDLeftList)):
                if k == j:
                    continue
                isIn = True
                for l in FDLeftList[k]:
                    if l not in thirdNFTables[thirdNFTables.index(i)][0]:
                        isIn = False
                        break
                if isIn == False:
                    continue
                for l in FDRightList[k]:
                    if l not in thirdNFTables[thirdNFTables.index(i)][0]:
                        isIn = False
                        break
                if isIn == False:
                    continue
                if primaryKeysAfter3nf[thirdNFTables.index(i)] != FDLeftList[k]:
                    continue
                isIn = True
                for l in FDLeftList[j]:
                    if l not in FDRightList[k]:
                        isIn = False
                        break
                if isIn == False:
                    continue
                thirdNFTables.append([])
                primaryKeysAfter3nf.append([])
                for l in range(len(thirdNFTables[thirdNFTables.index(i)])):
                    thirdNFTables[appendCount].append([])
                for l in FDLeftList[j]:
                    indexToAdd = thirdNFTables[thirdNFTables.index(i)][0].index(l)
                    primaryKeysAfter3nf[appendCount].append(l)
                    for m in range(len(thirdNFTables[appendCount])):
                        thirdNFTables[appendCount][m].append(thirdNFTables[thirdNFTables.index(i)][m][indexToAdd])
                for l in FDRightList[j]:
                    indexToAdd = thirdNFTables[thirdNFTables.index(i)][0].index(l)
                    for m in range(len(thirdNFTables[appendCount])):
                        pop3nf = secondNFTables[thirdNFTables.index(i)][m].pop(indexToAdd)
                        thirdNFTables[appendCount][m].append(pop3nf)
                appendCount += 1

    print('----------')
    print('bcnf:')
    for i in thirdNFTables:
        print('----------')
        for j in i:
            print(j)
        print('Primary keys:', primaryKeysAfter3nf[thirdNFTables.index(i)])
```

The code for BCNF is very similar to 3nf except it does not check for the right side of functional dependencies being prime attributes of the table.

<u>Fourth Normal Form:</u>

```
365 # 4nf --------------------------------------------------
366
367 if normalFormChoice == '4' or normalFormChoice == '5':
368     fourthNFTables = thirdNFTables.copy()
369     primaryKeysAfter4nf = primaryKeysAfter3nf.copy()
370     appendCount = len(thirdNFTables)
371
372     # Inputs and parses multivalued dependencies.
373     mvdLeftList = []
374     mvdRightList = []
375     mvd = ''
376     while mvd != 'done':
377         mvd = input("Multivalued Dependency (\'done\' if done): ")
378         if mvd == 'done':
379             break
380         splitmvd = mvd.split(' ->> ')
381         if '{' in splitmvd[0] and '}' in splitmvd[0]:
382             mvdLeftList.append(splitmvd[0][splitmvd[0].index('{')+1:splitmvd[0].index('}')].split(', '))
383         else:
384             mvdLeftList.append(splitmvd[0].split(', '))
385         mvdRightList.append(splitmvd[1].split('|'))
```

Lines 367 to 385 of 4nf input multivalued dependencies for the table originally chosen for normalization. It parses the input by splitting into two lists (one for the left side of the dependencies and one for the right).

```python
387     # Checks tables for presence of all attributes of a dependency and splits table while removing redundancies.
388     for i in range(len(fourthNFTables)):
389         for j in range(len(mvdLeftList)):
390             isIn = True
391             for k in mvdLeftList[j]:
392                 if k not in fourthNFTables[i][0]:
393                     isIn = False
394                     break
395             if isIn == False:
396                 continue
397             for k in mvdRightList[j]:
398                 if k not in fourthNFTables[i][0]:
399                     isIn = False
400                     break
401             if isIn == False:
402                 continue
403             for k in mvdRightList[j]:
404                 fourthNFTables.append([])
405                 primaryKeysAfter4nf.append([])
406                 for l in range(len(fourthNFTables[i])):
407                     fourthNFTables[appendCount].append([])
408                 for l in mvdLeftList[j]:
409                     indexToAdd = fourthNFTables[i][0].index(l)
410                     primaryKeysAfter4nf[appendCount].append(l)
411                     for m in range(len(fourthNFTables[appendCount])):
412                         fourthNFTables[appendCount][m].append(fourthNFTables[i][m][indexToAdd])
413                 indexToAdd = fourthNFTables[i][0].index(k)
414                 primaryKeysAfter4nf[appendCount].append(k)
415                 for l in range(len(fourthNFTables[appendCount])):
416                     pop4nf = fourthNFTables[i][l].pop(indexToAdd)
417                     fourthNFTables[appendCount][l].append(pop4nf)
418                 removeRedundancyFromAppend = []
419                 [removeRedundancyFromAppend.append(x) for x in fourthNFTables[appendCount] if x not in removeRedundancyFromAppend]
420                 fourthNFTables.pop(appendCount)
421                 fourthNFTables.append(removeRedundancyFromAppend)
422                 if fourthNFTables[appendCount][0] == originalPKs:
423                     fourthNFTables.pop(appendCount)
424                     primaryKeysAfter4nf.pop(appendCount)
425                 else:
426                     appendCount += 1
427             if fourthNFTables[i][0] == primaryKeysAfter4nf[i]:
428                 fourthNFTables.pop(i)
429                 primaryKeysAfter4nf.pop(i)
430                 appendCount -= 1
```

Lines 387 to 430 then loop through the multivalued dependency lists to check which tables contain all attributes of a given dependency and splits the tables to make sure there are no move mvds with the '|' symbol that are valid for a table.

```python
    print('----------')
    print('4nf:')
    for i in fourthNFTables:
        print('----------')
        for j in i:
            print(j)
        print('Primary keys:', primaryKeysAfter4nf[fourthNFTables.index(i)])
```

The result of 4nf is then printed in an orderly manner.

Fifth Normal Form:

```
# 5nf ---------------------------------------------------

if normalFormChoice == '5':
    fifthNFTables = fourthNFTables.copy()
    primaryKeysAfter5nf = primaryKeysAfter4nf.copy()
    tablesToPop = []

    for i in range(len(fourthNFTables)):
        matchFound = False
        possibleCommonAttributes = []
        for r in range(1,len(fourthNFTables[i][0]) - 1):
            possibleCommonAttributes.extend(itertools.combinations(fourthNFTables[i][0], r))
        for j in possibleCommonAttributes:
            remainingAttributes = []
            for k in fourthNFTables[i][0]:
                if k not in j:
                    remainingAttributes.append(k)
            possibleRemainingLeft = []
            for r in range(1,len(remainingAttributes)):
                possibleRemainingLeft.extend(itertools.combinations(remainingAttributes, r))
            for k in possibleRemainingLeft:
                possibleRemainingRight = []
                for l in fourthNFTables[i][0]:
                    if l not in k and l not in j:
                        possibleRemainingRight.append(l)
                # populates left table and right table, then check if union yields original table.
                leftTable = []
                rightTable = []
                for l in range(len(fourthNFTables[i])):
                    leftTable.append([])
                    rightTable.append([])
                for l in j:
                    indexToAdd = fourthNFTables[i][0].index(l)
                    for m in range(len(leftTable)):
                        leftTable[m].append(fourthNFTables[i][m][indexToAdd])
                        rightTable[m].append(fourthNFTables[i][m][indexToAdd])
                for l in k:
                    indexToAdd = fourthNFTables[i][0].index(l)
                    for m in range(len(leftTable)):
                        leftTable[m].append(fourthNFTables[i][m][indexToAdd])
                for l in possibleRemainingRight:
                    indexToAdd = fourthNFTables[i][0].index(l)
                    for m in range(len(rightTable)):
                        rightTable[m].append(fourthNFTables[i][m][indexToAdd])
                removeRedundancyFromAppend = []
                [removeRedundancyFromAppend.append(x) for x in leftTable if x not in removeRedundancyFromAppend]
                leftTable = removeRedundancyFromAppend.copy()
                removeRedundancyFromAppend = []
                [removeRedundancyFromAppend.append(x) for x in rightTable if x not in removeRedundancyFromAppend]
                rightTable = removeRedundancyFromAppend.copy()
                newTable = []
                appendCount2 = 0
                commonAttributesStart = len(j)
                for l in leftTable:
                    for m in rightTable:
                        if l[:commonAttributesStart] == m[:commonAttributesStart]:
                            newTable.append([])
                            newTable[appendCount2].extend(l)
                            newTable[appendCount2].extend(m[commonAttributesStart:])
                            appendCount2 += 1
                if newTable == fourthNFTables[i]:
                    matchFound = True
                    break
            if matchFound == True:
                break
        fifthNFTables.append(leftTable)
        fifthNFTables.append(rightTable)
        leftpks = []
        rightpks = []
        for j in primaryKeysAfter5nf[i]:
            if j in leftTable[0]:
                leftpks.append(j)
            if j in rightTable[0]:
                rightpks.append(j)
        primaryKeysAfter5nf.append(leftpks)
        primaryKeysAfter5nf.append(rightpks)
        if matchFound == True:
            tablesToPop.append(i)

    for i in tablesToPop:
        fifthNFTables.pop(i)
        primaryKeysAfter5nf.pop(i)
```

The code for 5nf finds all possible splits of each table by brute force. It then performs the natural join of each split and checks if the resulting table is equal to the original relation. If it is, it keeps the split.

```
    print('----------')
    print('5nf:')
    for i in fifthNFTables:
        print('----------')
        for j in i:
            print(j)
        print('Primary keys:', primaryKeysAfter5nf[fifthNFTables.index(i)])
```

If then prints the results of the 5nf normalization in an orderly manner.