IA160-3-FY Computer Programming

Project 2 Assignment

Magic Square Puzzle Board Validator

Lecturer: Dr. Ian Mothersole

Student Number: 1902160

Word count: 2196

Contents:

Introduction 2

Solution Explanation 2

`sqr_grid()` 2

`checkvalid()` 2

File I/O – Open and Read 3

`rowsize` 3

`sumofrcd` 4

Rows, Columns and Diagonals 4

Repetition 5

Valid Integers 5

Overall Validity, File I/O - Write and Close 5

Testing 5

Conclusion 13

Appendix 14

Introduction

The task was to create a python script that could validate a supplied "magic square puzzle board". The file name would be specified by the user and would be opened by the script in a `.txt` file format. A valid magic square will have various traits such as being an n^2 grid of

integers 1-n^2 with no repetition, and the integers will be arranged in such a way that all rows, columns, and diagonals all add up to the same number. Text files were provided with examples of valid magic square puzzles.
The assignment background states that if a grid is found to be valid, "it should be written into a new text file using the same format as the original input file" so although there were opportunities to potentially make invalid files valid, this would contradict the assignment and so in these situations, files were left unchanged. Any errors found in text files that meant it was not usable as a magic square grid were to be output to the user. Valid files were to be written to a new file with the prefix "VALID_". Use of functions was encouraged.

Solution Explanation

`sqr_grid()`

The `sqr_grid()` function takes one parameter and returns true if it is a square number. It checks this by raising the parameter to the power of 0.5 (which is equal to the square root) and dividing the result by an integer of itself. If the square root is a round number the result will be a float, for example 3.0. When this is divided by the integer 3 the result is 1 and the function returns `True` because if the square root is a round number then the number must be a square number. If the square root is not a round number, dividing by an integer of itself will not be equal to 1 and the function will return `False` . This is only used with the parameter `len(u)` which is equal to the length of the number of values in the text file.

`checkvalid()`

This function takes two parameters. The first parameter `dictionary` is the dictionary that contains either all the row values, the column values, or the diagonal values. The second will be a string: "row", "column" or "diagonal", used for string formatting purposes. For each key in `dictionary` the length of its value is checked to make sure that it has the right number of integers. All the values are then added together and it is checked that this total is equal to a variable called `sumofrcd` . `sumofrcd` (sum of rcd) is equal to the integer that each row, column and diagonal should add up to, therefore if each key in each dictionary adds up to `sumofrcd` then they all add up to the correct total. The function continues to loop through the dictionary until all of the keys have been checked for both correct length and sum. If any of the keys are the wrong length or total, the function will return `False`, and will return `True` otherwise.

File I/O – Open and Read

When the script is executed, the user is prompted to enter the name of the text file they wish to open which is saved to a variable called `filename` . `try` and `except` statements are

used to stop the script from crashing if an invalid filename is entered, and `filename` is checked to see if the string contains ".txt". If it does not then ".txt" is concatenated onto the string. This is then used as the first argument of the `open()` function. If no matching file is found, the user will be prompted again to enter the file name.
Once a valid file name is found, the file is opened in the `file` variable and its contents printed for the user to see. The `.seek()` method is used to reset the current file position to zero after printing `file.read()` for the user. Then a while loop containing a for loop is used to read the file line by line, split each line into a list of strings, and then add each string to a list in the `rows` dictionary as an integer. `try` and `except` statements are used to make sure that the script does not crash if there are any non-integer characters. In the case of an exception, the character is appended as a string. Although the strings could easily be omitted at this stage, potentially making the grid valid where it would have otherwise been invalid, this is one of those situations mentioned in the introduction that would contradict the assignment.
The `filesize` variable is equal to the how many lines there are in the file, this is so that if there is a blank line in the file, the script can check if this is the end of the file or just a blank line to be ignored before continuing. If more than one blank line is found (the end of the file), the `emptyline` variable is set to true and the file will be found invalid when the final checks are carried out.

`rowsize`

In a correctly sized grid, *n* x *n*; the total number of values is equal to *n* multiplied by *n,* or n^2, therefore *n* is equal to the square root of the number of values. The variable `rowsize` is equal to the `round()` of the square root of the length of `u` , which is equal to `file.read().split()` , which is a list of all the values in the grid. In short, $u = n^2$ and `rowsize` = *n*. The `round()` function in `rowsize` serves a dual purpose; it converts the float type result of the square root into an integer for string formatting purposes, and if the grid is not a correct size, I.e. if `u` is not equal to a square number, `rowsize` will be rounded to the closest number, allowing the script to differentiate between say, a 5x5 grid with up to 5 extra values, and a 6x6 grid that is missing up to 5 values.

`Sumofrcd`

Equation 1 shows a formula for calculating the total of each row, column, or diagonal in a valid *n* x *n* magic square that was reverse engineered from the known totals of grids of size 3 x 3

through 6 x 6. In a 4 x 4 grid, all rows, columns and diagonals should add up to 34. Through trial and error it was noted that 34 / 4 = 8.5 and 8.5 x 2 = 17 and $4^2 + 1 = 17$. Which led to the formula $T = (n^2 + 1) / 2 \times n$ where T is the total of each row etc. It worked for all known grid sizes so it was simplified to what can be seen in *Equation 1*. The variable `sumofrcd` is equal to the integer of the result of *Equation 1* where *n* is substituted with the variable `rowsize`.

$$\frac{n(n^2+1)}{2}$$

Rows, Columns and Diagonals

Once the size of the grid has been estimated, the script loops through the rows dictionary and takes the first index of each row and adds them to the `1` key in the column dictionary, it then adds the second index of each row to the `2` key, and so on until all of the columns are in the dictionary. A diagonal dictionary is then created via a similar loop, but with two hard coded keys in the `diag` dictionary: `1` and `2`. This is because for any given size of grid, there will only ever be two diagonals.

Dictionaries of lists of integers were chosen as the most suitable data structure as, once stored, a whole row, column, or diagonal can be referenced at once, without having to search the grid again. Also, with the values stored as integers, arithmetic operations can be performed easily without typecasting. An example of this, is adding all values in a column to see if it adds up to the correct value. Also, with the keys being numbers 1 to *n*, the keys can be used in string formatting to show which row/column/diagonal the output is referring to.

Repetition

The text file is checked for repeated values in a single line by checking if `u` (the list of all values) is the same length as `set(u)`. Because converting to a set will remove duplicate values, if u is the same length after being converted to a set, then there are no duplicate values. If the `set(u)` is not the same length as u then it must be shorter and there must be duplicate values. In this case a for loop is used to find which values are repeated and adds them to a dictionary where the number/character being repeated is the key, and the number of repetitions is the value. Both the keys and values can be used in string formatting and output to the user.

Valid Integers

Values are checked to be within the range of 1 to n^2 inclusive. This is accomplished by iterating through the values in the `rows` dictionary and making sure that each integer does not exceed the length of `u` (equal to n^2) and is not < 1. `try` and `except` statements are used here to catch any non-integer characters. Although non-integer characters would have already been processed when adding them to the `rows` dictionary, this is the point at which they are acknowledged and output to the user as a reason for invalidity.

Overall Validity, File I/O - Write and Close

An overall validity check is carried out in two stages. Firstly, the `checkvalid()` function is used on the `rows`, `colms` and `diag` dictionaries in sequence within an `if` statement. If all functions return `True`, then all rows, columns, and diagonals must each add up to the correct number, and a single variable `rcd` is set to `True` . This is then combined in another `if` statement with all the previous checks. If this check is true than the file is known to be valid, is written to a new file with the prefix "VALID_", and both files are closed. If any part of the final check is `False` then the user is informed that the file is not suitable.

Testing

The script was tested thoroughly by opening different text files. One set of valid files and one set that were made invalid in different ways. The invalid files were suffixed with "-b". For example: 3x3-b was the same as 3x3 except for some invalid characters added at the end of rows 2 and 3.

Here are two of the outputs from testing all of the valid files (3x3 to 6x6):

3x3:

```
Please enter the name of the text file you wish to open, for example '3x3' or
'4x4.txt': 3x3
```

```
8     1     6

3     5     7

4     9     2


No values are repeated.

There are a square number of values in the text file (9).

Considering the number of values; This should be a 3x3 grid.

All integers in grid are valid.

Estimated correct line total: 15

All rows add up to 15

All rows have the correct number of values (3)

All columns add up to 15

All columns have the correct number of values (3)

All diagonals add up to 15

All diagonals have the correct number of values (3)

File validated successfully!

Valid File Saved!
```

6x6:

```
Please enter the name of the text file you wish to open, for example '3x3' or
'4x4.txt': 6x6


6     32    3     34    35    1

7     11    27    28    8     30

19    14    16    15    23    24

18    20    22    21    17    13

25    29    10    9     26    12

36    5     33    4     2     31
```

```
No values are repeated.

There are a square number of values in the text file (36).

Considering the number of values; This should be a 6x6 grid.

All integers in grid are valid.

Estimated correct line total: 111

All rows add up to 111

All rows have the correct number of values (6)

All columns add up to 111

All columns have the correct number of values (6)

All diagonals add up to 111

All diagonals have the correct number of values (6)

File validated successfully!

Valid File Saved!
```

Here is a selection of outputs from intentionally invalidated files, beginning with the aforementioned 3x3-b. Note that anomalies are suffixed with the warning variable which prints as "". This is to highlight them for the user:

```
Please enter the name of the text file you wish to open, for example '3x3' or
'4x4.txt': 3x3-b


8     1     6

3     5     7     k

4     9     2     k


k is repeated 2 times.

Some values are repeated. CHECK FAILED.    <--- !!!

There are NOT a square number of values in the text file.    <--- !!!

Invalid character in text file: k    <--- !!!
```

```
Invalid character in text file: k    <--- !!!

Estimated correct line total: 15

Row 2 has 1 extra value.

Row 3 has 1 extra value.

All rows add up to 15

Not all rows have the correct number of values (3)    <--- !!!

File not suitable.
```

4x4-b with many repeated values:

```
Please enter the name of the text file you wish to open, for example '3x3' or
'4x4.txt': 4x4-b


1     1     1     1

2     2     2     2

3     3     3     3

4     5     6     7

1 is repeated 4 times.

2 is repeated 4 times.

3 is repeated 4 times.

Some values are repeated. CHECK FAILED.    <--- !!!

There are a square number of values in the text file (16).

Considering the number of values; This should be a 4x4 grid.

All integers in grid are valid.

Estimated correct line total: 34

Row 1 adds up to 4 . This is wrong.

Row 2 adds up to 8 . This is wrong.

Row 3 adds up to 12 . This is wrong.

Row 4 adds up to 22 . This is wrong.

Not all rows add up to 34 . CHECK FAILED.    <--- !!!
```

```
All rows have the correct number of values (4)

File not suitable.
```

4x4-b valid except for an empty line:

```
Please enter the name of the text file you wish to open, for example '3x3' or
'4x4.txt': 4x4-b


4     14    15    1

9     7     6     12


5     11    10    8

16    2     3     13

Blank lines removed for validation purposes but this file will NOT be valid.

Please remove blank lines.    <--- !!!

No values are repeated.

There are a square number of values in the text file (16).

Considering the number of values; This should be a 4x4 grid.

All integers in grid are valid.

Estimated correct line total: 34

All rows add up to 34

All rows have the correct number of values (4)

All columns add up to 34

All columns have the correct number of values (4)

All diagonals add up to 34

All diagonals have the correct number of values (4)

File not suitable.
```

5x5-b with an extra value on row 4, and two extra rows with 4 missing values (The script still recognised it as a 5x5 grid due to the number of values):

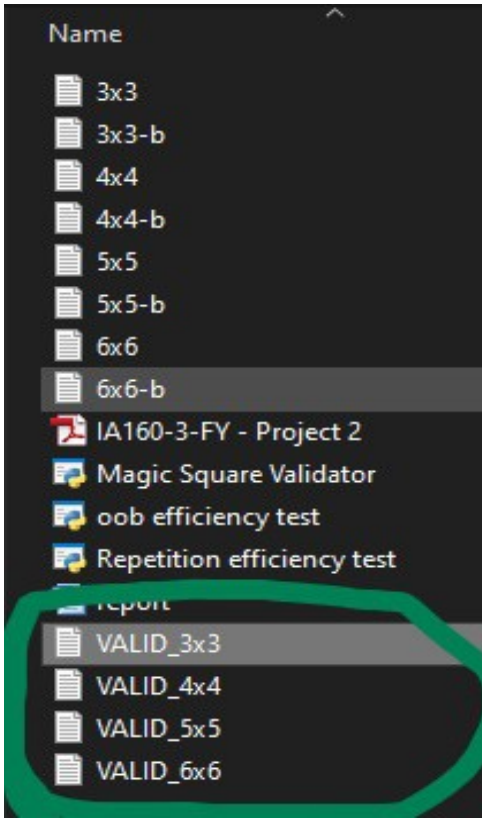
```
Please enter the name of the text file you wish to open, for example '3x3' or
'4x4.txt': 5x5-b


23    6     19    2     15

4     12    25    8     16

10    18    1     14    22

11    24    7     20    3     8

17    5     13    21    9

23

87

Error while separating column values, this is likely due to incorrect text
formatting.

Error while separating diagonal values, this is likely due to incorrect text
formatting.

23 is repeated 2 times.

8 is repeated 2 times.

Some values are repeated. CHECK FAILED.    <--- !!!

There are NOT a square number of values in the text file.    <--- !!!

Invalid integer in text file:  87

Estimated correct line total: 65

Row 4 has 1 extra value.

Row 4 adds up to 73 . This is wrong.

Row 6 is missing 4 values.

Row 6 adds up to 23 . This is wrong.

Row 7 is missing 4 values.

Row 7 adds up to 87 . This is wrong.

Not all rows add up to 65 . CHECK FAILED.    <--- !!!

Not all rows have the correct number of values (5)    <--- !!!

File not suitable.
```

6x6-b with an extra row of repeated values that add up to the same as the other rows:

```
Please enter the name of the text file you wish to open, for example '3x3' or
'4x4.txt': 6x6-b


6     32    3     34    35    1

7     11    27    28    8     30

19    14    16    15    23    24

18    20    22    21    17    13

25    29    10    9     26    12

36    5     33    4     2     31

40    20    10    30    5     6

Error while separating column values, this is likely due to incorrect text formatting.

Error while separating diagonal values, this is likely due to incorrect text formatting.

6 is repeated 2 times.

30 is repeated 2 times.

20 is repeated 2 times.

10 is repeated 2 times.

5 is repeated 2 times.

Some values are repeated. CHECK FAILED.    <--- !!!

There are NOT a square number of values in the text file.    <--- !!!

All integers in grid are valid.

Estimated correct line total: 111

All rows add up to 111

All rows have the correct number of values (6)

Column 1 has 1 extra value.

Column 1 adds up to 151 . This is wrong.

Column 2 has 1 extra value.

Column 2 adds up to 131 . This is wrong.
```

```
Column 3 has 1 extra value.

Column 3 adds up to 121 . This is wrong.

Column 4 has 1 extra value.

Column 4 adds up to 141 . This is wrong.

Column 5 has 1 extra value.

Column 5 adds up to 116 . This is wrong.

Column 6 has 1 extra value.

Column 6 adds up to 117 . This is wrong.

Not all columns add up to 111 . CHECK FAILED.    <--- !!!

Not all columns have the correct number of values (6)    <--- !!!

File not suitable.
```

It is important that the files write correctly so the directory was checked for the written files with "VALID_" prefix, and files were opened to ensure they were correctly formatted:



The efficiency of the script was also tested. One such test was on the block that checks for repetition in the grid. Originally the code was as follows:

```
repeated = {}
for x in u:
      if u.count(x) > 1:
            repeated[x] = u.count(x)
```

The perceived issue with this code was that if there was repetition in the code, every occurrence of that integer would just be overwriting the first. This still works, but is inefficient. So it was considered to add in the line "`if x not in repeated:`" like so:

```
repeated = {}
for x in u:
      if x not in repeated:
            if u.count(x) > 1:
                  repeated[x] = u.count(x)
```

This would skip integers that had already been added to the dictionary of repeated integers, but there were concerns that in the case that there were no repeated integers this would be less efficient as it is just an extra if statement to process. A short script was written to test the process time of both sections in isolation.

```
Current - repeated integer: 1.0
not_in - repeated integer: 0.0
current - no repetition: 0.9375
not_in - no repetition: 0.9375
```

The results above (lower is faster) show that having "`if x not in repeated:`" is much quicker if there are repeated integers, and around the same speed if there is no repetition. For this reason, the `if` statement was added into the script.

Conclusion

The script is a capable magic square validator with exception and input handling that allows for a variety of errors without crashing. The introduction mentioned that errors were to be output to the user. No mention was made of outputting when the checks have been successful, for this reason, successful checks were kept more general i.e. "`All integers in grid are valid.`" Whereas invalid checks were more specific i.e "`Row 3 has 1 extra value.`". The script is well tested for both efficiency and errors. It enables the user to specify the name of a text file and adds the file extension if it is missing. All of the traits that define a magic square are tested for. Valid grids are written into a new text file using the same format as the original input file. The functions are efficient and save repetition.

Appendix

```
def sqr_grid(v): # a function to determine if a number is square (returns True if square)
    root = v ** 0.5
    if root / int(root) == 1:
        return True
    elif root / int(root) != 1:
        return False


# Perform check to see if there are the same number of values on each line
# and if those values add up to the correct number.
def checkvalid(dictionary):
    if dictionary == rows:
        name = "row"
    elif dictionary == colms:
        name = "column"
    elif dictionary == diag:
        name = "diagonal"
    lenvalid = True
    sumvalid = True
    root = int(len(u) ** 0.5)
    for x in dictionary:
        suf = ""
        linelen = len(dictionary[x])


        if linelen > (root + 1) or linelen < (root - 1):
            suf = "s"
        if linelen > root:
```

```
            print("\n{} {} has {} extra value{}.".format(name.title(), x, (linelen - root),
suf) )
            lenvalid = False
        elif linelen < root:
            print("\n{} {} is missing {} value{}.".format(name.title(), x, (root - linelen),
suf) )
            lenvalid = False


        totalline = 0
        for n in dictionary[x]:
            try:
                totalline += int(n)
            except:
                continue
        if totalline != sumofrcd:
            print(name.capitalize(), x, "adds up to", totalline, ". This is wrong.")
            sumvalid = False


    if sumvalid == True:
        print("\nAll {}s add up to".format(name), sumofrcd)
    elif sumvalid == False:
        print("Not all {}s add up to".format(name), sumofrcd, ". CHECK FAILED.", warning)


    if lenvalid == True:
        print("\nAll {}s have the correct number of values ({})".format(name, rowsize))
    elif lenvalid == False:
        print("\nNot all {}s have the correct number of values ({})".format(name, rowsize),
warning)
```

```
    if lenvalid != True or sumvalid != True:
        return False
    return True


warning = "   <--- !!!"


while True:
    while True:
        try:
            filename = input("\nPlease enter the name of the text file you wish to open, for
example '3x3' or '4x4.txt': ")
            print("\n")
            if ".txt" not in filename:
                filename += ".txt"
            file = open(filename, "r")
            break
        except:
            print("That is not a valid file name.")
    print(file.read())
    file.seek(0)
    rows = {}
    empty = 0
    i = 1
    filesize = len(file.readlines())
    file.seek(0)
    while True:
        line = file.readline().split()
        if line != []:
            rows[i] = []
```

```
            for x in line:
                try:
                    rows[i].append(int(x))# add each value in the line to a dictionary of
rows as an integer
                except:
                    rows[i].append(x)
            i+= 1
        elif line == []:
            empty += 1
            if len(rows) < filesize - empty:
                continue
            elif len(rows) >= filesize - empty:
                break


    emptyline = False
    if empty > 1:
        print("\nBlank lines removed for validation purposes but this file will NOT be
valid. \nPlease remove blank lines.", warning)
        emptyline = True


    file.seek(0)
    u = file.read().split() # u is a list of all values in the grid for testing number of
values and for repeat values
    rowsize = round(len(u) ** 0.5) #Determines proper row length by getting the square root
of number of values




    # Find what each row/column/diagonal should add up to with a formula that takes the
total number
    # of values (Row size squared) and the row size (square root of number of values)
```

```
    # and results in the only integer that each R/C/D can equal if you rearrange all the
values so that they all

    # add up to the same integer

    sumofrcd = int(((rowsize ** 2) +1) / 2 * rowsize) #sum of rcd = sum of Rows, Columns,
Diagonals




    # Make a dictionary of columns from the rows

    colms = {}

    try:

        for k in range(0,len(rows)):

            for x in range(1,len(rows) + 1):

                try:

                    colms[k + 1].append(rows[x][k]) # Add the 'k'th index from row[x] to the
'k'th column

                except:

                    colms[k + 1] = [rows[x][k]]

    except:

        print("Error while separating column values, this is likely due to incorrect text
formatting.")



    # Make dictionary of diagonals from the rows

    diag = {1:[], 2:[]} # Two keys are hardcoded as this will be the same for any size grid

    i = 0

    try:

        for x in rows:

            diag[1].append(rows[x][i])

            i+=1

        for x in rows:

            i -= 1
```

```
            diag[2].append(rows[x][i])
    except:
        print("Error while separating diagonal values, this is likely due to incorrect text formatting.")


    if len(u) == len(set(u)): # check for repeated values
        repeats = False
        print("\nNo values are repeated.")
    else:
        repeated = {}
        for x in u:
            if x not in repeated:
                if u.count(x) > 1:
                    repeated[x] = u.count(x)
        for n in repeated:
            print(n, "is repeated", repeated[n], "times.")
        repeats = True
    if repeats == True:
        print("Some values are repeated. CHECK FAILED.", warning)




    if sqr_grid(len(u)) == True: #if there are a square number of values, then we know there is only one suitable order (length of rows/columns)
        print("\nThere are a square number of values in the text file ({}). \n".format(len(u)))
        print("Considering the number of values; This should be a " + str(rowsize) + "x" + str(rowsize) + " grid.")
    else:
        print("\nThere are NOT a square number of values in the text file.", warning)
```

```
    oob = False #out of bounds
    for x in rows: # check if any integers excede total number of values or are invalid
        for n in rows[x]:
            try:
                if n > len(u) or n < 1:
                    print("\nInvalid integer in text file: ", n)
                    oob = True
            except:
                print("\nInvalid character in text file:", n, warning)
                oob = True
    if oob == False:
        print("\nAll integers in grid are valid.")


    print("\nEstimated correct line total:", sumofrcd)
    # check that all rows, columns and diagonals add up to the right number and have the
right length
    rcd = False
    if  checkvalid(rows) == True and checkvalid(colms) == True and checkvalid(diag) == True:
        rcd = True


    # Overall check/Results
    if emptyline == False and sqr_grid(len(u)) == True and oob == False and repeats == False
and rcd == True:
        file.seek(0)
        print("\nFile validated successfully!")
        valid = open("VALID_" + filename, 'w')
        valid.write(file.read())
```

```
    valid.close()
    file.close()
    print("Valid File Saved!")
else:
    print("\nFile not suitable.")
```