UNIVERSITY OF SASKATCHEWAN
Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 1

Getting Started – Solutions

Date Due: January 19, 2018, 10pm **Total Marks: 50**

Question 1 (10 points):

Purpose: To build a program and test it. To get warmed up with Python, in case you are using it for the first time.

Degree of Difficulty: Moderate. Don't leave this to the last minute. The basic functionality is easy. There are aspects of the problem that you won't appreciate until you start testing.

A *Magic Square* is an arrangement of numbers in a square, so that every row, column, and diagonal add up to the same value. Below are two squares, but only one of them is a Magic Square.

8	1	6
3	5	7
4	9	2

1	9	6
5	3	7
4	8	2

The square on the left is a 3×3 magic square, whose rows, columns, and diagonals all sum to 15. On the right, is a 3×3 square of numbers whose rows columns and diagonals don't have the same sum.

There are magic squares of all sizes, but we'll be concerned with 3×3 squares, and checking if a given arrangement of 9 numbers is a magic square or not.

Definition: A 3×3 magic square is formally defined by the following three criteria:

- It contains the integers 1 through 9 inclusively.
- Every integer in the range 1 through 9 appears exactly once.
- Every row, column, and diagonal sums to 15.

In this question you will implement a program that does the following:

- It asks the user for a sequence of 9 numbers from the console. The order of the numbers is important, as the rows of the grid use this order. **For simplicity, assume that the user will type integers on the console. For this question, you don't have worry about what to do if the user types anything other than integers.**
- It checks whether the sequence of integers is a magic square or not. Your program should display the message "yes" if it's magic, or "no" if it's not.

It's very important to point out that **you are not being asked to construct a magic square; only to check if a square is magic or not.**

What to Hand In

- Your implementation of the program: `a1q1.py`.
- A text document called `a1q1_demo.txt`, showing at least six (6) demonstrations of your program working on 3 examples that are true magic squares, and 3 examples that are not magic squares. This is a demonstration, not testing.
- If you wrote a test script, hand that in too, calling it `a1q1_testing.txt`

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your program works.
- 5 marks: Your program is well-documented.

Solution: In the Solutions folder, you will find a design document for this question, named `MagicSquareDD.txt`, which is a very thorough example. The problem is explained, and functions are described in terms of inputs, outputs, and purpose. Every function has test cases listed, and many (but not all) of the functions have pseudocode algorithms.

The Solutions folder also contains the implementation of this application in the file `a1q1.py`. The functions are thoroughly documented, and correspond with the design.

This example is extreme. Students are not expected to have been so thorough in Assignment 1, but it serves as an example of design that students can strive for in the future.

Here are some things for students to observe in the solution:

- Documentation. Every function is documented with a doc-string. There are not many other comments, though.
- Use of lists. The program uses a list of lists to represent the square. It's possible to use a simple list, and also acceptable to use a numpy array. Using a list of lists makes some parts of the task easy, but others a bit trickier.
- Use of functions. The design broke the problem into several small functions, each of which can be tested independently. The value of this approach is worth the effort. It's far easier to test and debug a 5 line function than a 40 line script.
- Testing. A separate test script was provided. The test script and the implementation were implemented one function at a time. One function was written, and then thoroughly tested, before moving to the next function. The advantage to this is that you never need to guess where the errors are: they have to be in the function you are currently working on.

Notes for markers:

- Last year, the same question was given to students, but we also included a pseudo-code design document. This year, the design document was not given.
- Students were given freedom to implement the program in any way they wanted. There is no requirement for use of functions, or list-of-lists, or anything.
- Check the `a1q1_demo.txt` file for the required demonstrations, and check to see that the code plausibly produced the demonstration output. Run the code only if you need to.
- Give full marks if you think the program works, no matter how ugly or terrible the code is. You may however leave a comment such as "This code works, but it's not good code."
- To assess documentation, look for:
  - Student information at the top of the program.
  - Some kind of doc-string or comment for each function. It need not be as thorough as the model solution, but there has to be something.
  - Comments here and there when the code looks complicated.
  - Over commenting. Too many comments are not as bad as no comments at all, but just by a little.
- Students were encouraged to submit a test script if they made one. There are no marks for this, but it may help you decide if on the 5 marks for correctness.

Question 2 (10 points):

Purpose: To reflect on the work of programming for Question 1. To practice objectively assessing the quality of the software you write. To practice visualizing improvements, without implementing improvements.

Degree of Difficulty: Easy.

Answer the following questions about your experience implementing the program in Question 1. You may use point form, and informal language. Just comment on your perceptions; you do not have to give really deep answers. Be brief. These are not deep questions; a couple of sentences or so ought to do it.

1. (2 marks) Comment on your program's correctness. How confident are you that your program (or the functions that you completed) is correct? What new information (in addition to your current level of testing) would raise your confidence? How likely is it that your program might be incorrect in a way you do not currently recognize?
2. (2 marks) Comment on your program's efficiency. How confident are you that your program is is reasonably efficient? What facts or concepts did you use to estimate or quantify your program's efficiency?
3. (2 marks) Comment on your program's adaptability. For example, what if Assignment 2 asked you to write a program to check whether a 5×5 square was magic (bigger square with a larger sum, using the numbers 1 through 25)? How hard would it be to take your work in A1Q1, and revise it to handle squares of any size?
4. (2 marks) Comment on your program's robustness. Can you identify places where your program might behave badly, even though you've done your best to make it correct? You do not have to fix anything you mention here; it's just good to be aware.
5. (2 marks) How much time did you spend writing your program? Did it take longer or shorter than you expected? If anything surprised you about this task, explain why it surprised you.

You are not being asked to defend your program as being good in all these considerations. For example, if your program is not very robust, you can say that; you don't need to make it robust.

What to Hand In

Your answers to the above questions in a text file called `a1_reflections.txt` (PDF, rtf, docx or doc are acceptable). Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

Each answer is worth 2 marks. Full marks will be given for any answer that demonstrates thoughtful reflection. Grammar and spelling wont be graded, but practice your professional-level writing skills anyway.

Solution: There are no wrong answers, but marks will be deducted if your answers are not relevant to the issues at hand. Here are some comments, and the level of discussion we would expect.

1. The test script suggests that the code is correct, with high confidence. I don't think adding more tests would increase my confidence.
2. The 3×3 case is so small that almost any implementation might count as reasonably efficient. However, there are parts that could be implemented quite inefficiently. For example, it's possible to look through the square once for each of the numbers 1 through 9, which would take more time than the implementation that used a Boolean list. But using the list uses more memory. Again the 3×3 case is so small that it hardly matters. But these considerations certainly would matter if we were looking for 3000×3000 magic squares.
3. The program I wrote would take some effort to adapt to the 5×5 case. Not difficult work, but almost every function would need to be changed. There is certainly room for better adaptability here!
4. All the check functions are reasonably robust. Bad things would happen if the square were not really a square, but some other kind of list, e.g., a list of strings, or a list of the wrong dimensions. The function `get_square()` is quite vulnerable to user error, and so it is not at all robust. If the user types too few numbers, the program will crash with a run-time error. If the user types non-numeric data, it will also crash.
5. Writing the design document took about 2 hours, including the test cases. The implementation took about an hour after that. I estimate it would have taken less time to write without a design document, but I would have taken more short-cuts, and I would not have had such neat functions to test. Debugging might have required more of my time, even if I spent less time writing code.

Notes for markers:

- This question is for students to practice thinking about these issues. There is no need for the answers to be positive. If the student knows their code is not very good, and says why, that's worth full marks here.
- Give zero marks for any part if no response was given, or if the response was completely irrelevant.
- Give one mark if the response seemed too superficial. A superficial response would be something like: "I'm sure my program is correct" without referring to test cases.
- CMPT145 students have not yet been taught anything formal about complexity analysis, so discussion on efficiency will be general and somewhat vague. That is perfectly acceptable for Assignment 1.

Question 3 (10 points):

Purpose: To build a program and test it. To get warmed up with Python, in case you are using it for the first time.

Degree of Difficulty: Moderate. Don't leave this to the last minute. The basic functionality is easy. There are aspects of the problem that you won't appreciate until you start testing.

A *Latin Square* is an arrangement of numbers 1 to N in a square, so that every row and column contain all the numbers from 1 to N. The order of the numbers in a row or column does not matter. Below are two squares, but only one of them is a Latin Square.

1	3	2
3	2	1
2	1	3

1	2	3
2	1	3
3	3	1

The square on the left is a Latin Square for the numbers 1,2,3, whose rows consist of the numbers 1,2,3 (in any order) and whose columns consist of the numbers 1,2,3 (in any order). On the right, is a 3×3 square of numbers that is *almost* a Latin Square, but the bottom row is missing the value 2, and the column on the right is also missing the value 2. There are Latin squares of all sizes.

Definition: A $N \times N$ Latin square is formally defined by the following criteria:

- Every row contains the numbers 1 through N exactly once, in any order.
- Every column contains the numbers 1 through N exactly once, in any order.

When the Latin square is $N \times N$ we say that it has "order N". The order tells us which numbers to look for, and how big the square is.

In this question you will implement a program that checks whether a $N \times N$ square of numbers is a Latin Square or not. Your program should work by reading input from the console, and sending an answer to the console, as in the following example:

- It reads a number N on a line by itself. This will be the order of a Latin square. The order must be a positive integer, e.g., $N > 0$.
- It reads N lines of N of numbers, i.e., it reads console input for a square of numbers.
- It checks whether the sequence of numbers is a Latin square or not. Your program should display the message "yes" if it satisfies the above criteria, or "no" if it does not.

What to Hand In

- Your implementation of the program: `a1q3.py`.
- A text document called `a1q3_demo.txt`, showing at least six (6) demonstrations of your program working on 3 examples that are true Latin squares, and 3 examples that are not Latin squares. This is a demonstration, not testing.
- If you wrote a test script, hand that in too, calling it `a1q3_testing.txt`

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your program works.
- 5 marks: Your program is well-documented.

Solution: A solution is provided in the Solutions folder.

In contrast to Question 1, this was not designed, and is very poorly documented. It's an example of code that a student might write without any planning or design.

This is an example of code that works in demonstration, but is poorly tested, and very poorly documented. This example might get 5/5 for correctness, but 0/5 for documentation.

Notes for markers:

- This exercise is intended to allow students to do whatever they feel like doing to solve it. It is completely acceptable to write a very simple program that gets the job done. It is also acceptable to borrow functions or designs from Q1.
- Students were given freedom to implement the program in any way they wanted. There is no requirement for use of functions, or list-of-lists, or anything.
- Check the `a1q3_demo.txt` file for the required demonstrations, and check to see that the code plausibly produced the demonstration output. Run the code only if you need to.
- Give full marks if you think the program works, no matter how ugly or terrible the code is. You may however leave a comment such as "This code works, but it's not good code."
- To assess documentation, look for:
  - Student information at the top of the program.
  - Some kind of doc-string or comment for each function. It need not be as thorough as the model solution, but there has to be something.
  - Comments here and there when the code looks complicated.
  - Over commenting. Too many comments are not as bad as no comments at all, but just by a little.
- Students were encouraged to submit a test script if they made one. There are no marks for this, but it may help you decide if on the 5 marks for correctness.

Question 4 (10 points):

Purpose: To reflect on the work of programming for Question 3. To practice objectively assessing the quality of the software you write. To practice visualizing improvements, without implementing improvements.

Degree of Difficulty: Easy.

Answer the following questions about your experience implementing the program in Question 3. You may use point form, and informal language. Just comment on your perceptions; you do not have to give really deep answers. Be brief. These are not deep questions; a couple of sentences or so ought to do it.

1. (2 marks) Comment on your program's correctness. How confident are you that your program (or the functions that you completed) is correct?
2. (2 marks) Comment on your program's efficiency. How confident are you that your program is is reasonably efficient?
3. (2 marks) Comment on your program's reusability. For example, did you re-use any code from a different project (maybe Q1)? How easy would it be to re-use any part of your program for another task?
4. (2 marks) Comment on your program's robustness. Can you identify places where your program might behave badly, even though you've done your best to make it correct? You do not have to fix anything you mention here.
5. (2 marks) How much time did you spend writing your program? Did it take longer or shorter than you expected? If anything surprised you about this task, explain why it surprised you.

You are not being asked to defend your program as being good in all these considerations. For example, if your program is not very robust, you can say that; you don't need to make it robust.

What to Hand In

Your answers to the above questions in the text file called `a1_reflections.txt` (PDF, rtf, docx or doc are acceptable). **This is the same document you used for Q2.** Please help the marker by clearly indicating the question number.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

Each answer is worth 2 marks. Full marks will be given for any answer that demonstrates thoughtful reflection. Grammar and spelling wont be graded, but practice your professional-level writing skills anyway.

Solution: Here are some responses to the questions, demonstrating the level of discussion we would expect. The comments are reflecting on the example solution provided for Question 3, which was probably correct, but poorly documented.

1. Using my UNIX skills, I tested it on 16 true Latin Squares, and 16 examples that were not Latin Squares. Every test passed, so I am fairly confident it is correct, when given numeric data.
2. If the input is a Latin Square, you have to look at each value. You can't get around that. If the program finds a row or column that does not satisfy the definition, it terminates early (`exit()`, so that's good. The program does not store a lot of extra data: only the input data. I would say it's well within the range of reasonable efficiency.
3. The program I wrote is undocumented and cryptic. It's a good example of a bad example of documentation! It could not be reused for much as it is.
4. The program will crash if the data is non-numeric, so it's not robust there. There's not much else that can go wrong unexpectedly.
5. I wrote my code in about 15 minutes. It's a good deal simpler than the task in Question 1, and I know what I'm doing (Latin Squares are a very useful test tube for many of the AI algorithms I have studied in my research). Since my aim was to demonstrate "throw away" quality, I took no care to document my ideas, nor did I make any effort to address any implementation goals. Even though I did not show my testing script, I did make 32 test cases and every time I modified my program, I tested all 32 cases. I may sometimes write throw-away code, but if the throw away code is not correct, you should throw it away before you use it!

Notes for markers:

- There are no "right answers" here. As long as the discussion is relevant, a generous mark is fine.
- There is no need for the answers to be positive. If the student knows their code is not very good, and says why, that's worth full marks here.
- Give zero marks for any part if no response was given, or if the response was completely irrelevant.
- Give one mark if the response seemed too superficial. A superficial response would be something like: "I'm sure my program is correct" without referring to test cases.
- CMPT145 students have not yet been taught anything formal about complexity analysis, so discussion on efficiency will be general and somewhat vague. That is perfectly acceptable for Assignment 1.

Question 5 (10 points):

Purpose: To work with the concept of references a bit more carefully. To debug a program that uses references incorrectly.

Degree of Difficulty: Moderate. If you understand references very well, this is not difficult.

In the file named `histogram-broken.py`, you'll find code to display a primitive histogram on the console based on integers typed on the console by the user. Here is an example of how it might work:

```
How many data values? 10
Enter integer data

1 2 3 4 5 4 3 2 3 4

 ----------- Histogram ----------------

1: *
2: **
3: ***
4: ***
5: *
```

The Histogram is supposed to print one * for each occurrence of the value in the input. Histograms are useful for visualizing the distribution of data. For example, from this histogram, we can see that 3 and 4 were the most common values in the input (there were three of each).

Unfortunately, the implementation in the file we've provided has several errors, as if written by a novice programmer who didn't take the time to design the algorithm carefully, and so did not quite have enough time to finish.

Your task is as follows:

1. Study the functions carefully. While there are some errors, there are parts of the functions that are very close to working. The purpose of this question is not to get you to invent anything tricky, but to get you to practice debugging.
2. Figure out what each part of the functions is supposed to do. The programmer left no helpful comments to assist your study. Sorry.
3. Add doc-strings appropriate to the functions. You might have to modify the doc-strings if you change the functions.
4. Fix the obvious errors, and then try to get the implementation working by debugging and testing.
5. Make a list briefly mentioning every change you make to the program. For example you might have a list that starts like this:

   ```
   1. Deleted line 7 of the original program.
   2. Changed function isNegative() to return a Boolean value instead of a string.
   ```

Hints: We created these functions by starting functions that worked perfectly, and then we added errors to it, similar to the errors novices might make. Most of the errors are related to the material we've covered in lecture so far, including review and the unit on references. Careful study should be enough to fix this program. You won't have to add much code at all. Just fix what's there.

What to hand in:

- The list of changes you made to the program, in the text file called `a1_reflections.txt` (PDF, rtf, docx or doc are acceptable). This is the same document you used earlier. Please help the marker by clearly indicating the question number.
- Hand in your working program in a file called `a1q5.py`.
- A text-document called `a1q5_demo.txt` that shows that your program working on a few examples. You may copy/paste to a text document from the console.
- If you wrote a test script, hand that in too, calling it `a1q5_testing.txt`

Evaluation

- 4 marks. Your list of changes includes the bugs we know about in the original program.
- 3 marks. Your program works.
- 3 marks. Your functions have appropriate doc-strings, and other appropriate documentation.

Solution: This debugging exercise was difficult.

Here are a list of the bugs in the given broken program:

- Line 10. `find_min_and_max()` tries to modify parameters, but this is ineffectual. Possible solution: return the min and max values instead.
- `counting()`
  - Line 20. Uses an incorrect formula to determine the size of the frequency array.
  - Line 18. Tries to use a function to calculate the min and max values, but this does not have any affect on the variables. Possible solution: make min and max parameters, and calculate the min and max outside the function.
  - Line 24. Uses an incorrect mapping from data value to index. Possible solution: delete the +1.
- Line 34. Displays the count of data values instead of the data values. Possible solution: use the min data value and the index for the frequency array to determine the data value.
- Line 38. Does not send a useful argument as a list to `get_data()`. Also, `get_data()` does not have a return statement, so it returns `None`, which is not a valid (or useful) data list. Possible solution: return a data list, and remove the useless parameter.
- Line 40. Does not send useful min max data values to the histogram function. Possible solution: Calculate the min and max data values in the main script, to inform all the functions.

Notes for markers:

- To assess the list of changes, you can compare the changes to the list above. Give 4/4 if more than half of the bugs were mentioned (as changes), and 2/4 if only about half of the bugs were mentioned. Be generous.
- It's okay for students to rewrite portions, but if they rewrote too much, give 2/4.
- To assess correctness, check the demonstration file. It was fairly easy to get the program to work with a small amount of data, but to handle negative data, the bugs in count() have to be fixed. If they didn't test negatives, you can be sure they didn't find all the bugs.
- Each function should have a doc-string or comments describing the purpose, inputs and outputs. If they did that only, it's worth 2/3. Give the extra point if they made an effort to document any of the trickier parts of the code.
- It's okay to be generous. Give 2s unless they really made good effort, or really made a weak effort.