FIT1045 Algorithms and programming in Python, S2-2021
Programming Assignment

Assessment value: 22% (10% for Part 1 + 12% for Part 2)

Due: Friday Week 6 (Part 1), Friday Week 11 (Part 2)

This assignment is designed to practice and assess your capabilities to solve complex algorithmic problems with Python AND your ability to verbally explain your solution. **It is an individual assignment meaning that you are supposed to solve it alone during your self-study time and are not allowed to share or copy your solution or any part of it.** Please ensure that you have read and understood the university's policies on plagiarism and collusion available at `http://www.monash.edu.au/students/policies/academic-integrity.html`. You will be required to agree to these policies when you submit your assignment.

The above implies that you are not allowed to post any of your code to any forum (including Ed). However, please do use Ed to clarify task requirements and to discuss other assignment-related questions that do not disclose parts of your solution. If you feel stuck and believe you really need more specific help, please approach your tutors. They won't solve the problems for you (and especially not debug your code) but can provide you with general feedback and tell you if you are on the right track. While parts of the assignment might be challenging, we hope that completing it will be a fruitful and engaging learning experience for you.

Marks, Submission, and Interviews

Each part of the assignment has a higher number of mark opportunities than the maximum mark for that part. This means that you don't have to complete all tasks to achieve the maximum mark.

To obtain marks for this assignment you have to submit two versions of a module file `sudoku.py` (based on provided template) via Moodle at the due dates for Part 1 and Part 2, respectively AND attend the corresponding assignment interviews during your lab class in the weeks after the submissions. **Not attending the interview will result in 0 marks for the corresponding part of the assignment.**

During the interviews you have to be able to explain your solution (otherwise any or all marks can be deducted). Thus, it is highly recommended that you keep your submission in a clean form with well decomposed and documented functions and only use programming constructs that you can explain. If you are unable to explain your solution, the maximum number of marks that you can achieve for the part of the assignment is capped to:

- 0 marks if you don't attend the interview or don't participate in it in a meaningful way
- 25% of the maximally attainable marks if you are unable to explain the Python language elements you use in your code (e.g., you use a range but do not know what it is)
- 49% of the maximally attainable marks if you are unable to explain the purpose of parts of your submission (e.g., you cannot explain what an input to a function means or what output is computed for it).
- 69% of the maximally attainable marks if you are unable to explain how your solution works (e.g., you are unable to verbalise the overall algorithmic strategy that is used in a part of your solution).

Do not enter your name or student id into the module file. Instead, rename your file to `sudoku_[id].py` where you replace `[id]` by your student id. Your module is not allowed to use imports except for the modules math, copy, and random.

Sudoku

Sudoku is a puzzle game for one player where one has to fill up a regular grid of fields (game board) with numbers. Typically a Sudoku board is 9 times 9 fields, but in this assignment we will write functions that can work in principle with arbitrary sized n times n boards as long as `n = k**2` for some integer `k > 1` (although for `k > 2` some functions in Part II may be too slow to use in practice).

Conceptually, the board is composed of `k`×`k` subgrids (each consisting of `k`×`k` fields). The objective of the player is to fill all fields with the numbers 1 to n (inclusive) such that

- no column of the grid contains the same number more than once
- now row of the grid contains the same number more than once
- none of the `k**2` subgrids contains the same number more than once

See `https://en.wikipedia.org/wiki/Sudoku` for more information.

In this assignment, we represent a Sudoku board by a `n`×`n` table where each entry is either a number from `1` to `n` (inclusive) or `0` representing that the corresponding field is still empty. For example, a small game board with `n=4` (and `k=2`) with four fields already filled could be defined as follows:

```
>>> small = [[0, 0, 1, 0],
...          [4, 0, 0, 0],
...          [0, 0, 0, 2],
...          [0, 3, 0, 0]]
```

An example for the typical 9x9 size is:

```
>>> big = [[0, 0, 0, 0, 0, 0, 0, 0, 0],
...        [4, 0, 0, 7, 8, 9, 0, 0, 0],
...        [7, 8, 0, 0, 0, 0, 0, 5, 6],
...        [0, 2, 0, 3, 6, 0, 8, 0, 0],
...        [0, 0, 5, 0, 0, 7, 0, 1, 0],
...        [8, 0, 0, 2, 0, 0, 0, 0, 5],
...        [0, 0, 1, 6, 4, 0, 9, 7, 0],
...        [0, 0, 0, 9, 0, 0, 0, 0, 0],
...        [0, 0, 0, 0, 3, 0, 0, 0, 2]]
```

Finally, an example of a giant board with 4x4 subgrids each of size 4x4 is:

```
>>> giant = [[ 0,  5,  0,  0,  0,  4,  0,  8,  0,  6,  0,  0,  0,  0,  9, 16],
...          [ 1,  0,  0,  0,  0,  0,  0, 13,  4,  0,  0,  7, 15,  0,  8,  0],
...          [13,  0,  0,  0,  0,  7,  3,  0,  0,  0,  0,  9,  5, 10,  0,  0],
...          [ 0, 11, 12, 15, 10,  0,  0,  0,  0,  0,  5,  0,  3,  4,  0, 13],
...          [15,  0,  1,  3,  0,  0,  7,  2,  0,  0,  0,  0,  0,  5,  0,  0],
...          [ 0,  0,  0, 12,  0,  3,  0,  5,  0, 11,  0, 14,  0,  0,  0,  9],
...          [ 4,  7,  0,  0,  0,  0,  0,  0, 12,  0, 15, 16,  0,  0,  0,  0],
...          [ 0,  0,  0,  0, 14,  0, 15,  0,  6,  9,  0,  0,  0,  0, 12,  0],
...          [ 3,  0, 15,  4,  0, 13, 14,  0,  0,  0,  0,  1,  0,  0,  7,  8],
...          [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  9, 10,  0,  0,  0,  0],
...          [11,  0, 16, 10,  0,  0,  0,  0,  0,  7,  0,  0,  0,  3,  5,  0],
...          [ 0,  0, 13,  0,  0,  0,  0,  0, 14,  0, 15, 16,  0,  9,  0,  1],
...          [ 9,  0,  2,  0,  0, 14,  0,  4,  8,  0,  0,  0,  0,  0,  0,  0],
...          [ 0, 14,  0,  0,  0,  0,  0, 10,  9,  0,  3,  0,  0,  0,  1,  7],
...          [ 8,  0,  0,  0, 16,  0,  0,  1,  2, 14, 11,  4,  0,  0,  0,  3],
...          [ 0,  0,  0,  1,  0,  0,  5,  0,  0, 16,  0,  6,  0, 12,  0,  0]]
```

Part 1 – Play Sudoku (max 10 marks)

A: Display Sudoku (5 marks)

The function `print_board(board)` from the template accepts as input parameter a game board (board) prints simply by using the built-in print function. This is very unsatisfactory because it doesn't align the columns of the board and leaves the subgrids invisible. Also it prints zeroes, which would be more appropriately displayed as spaces (as they represent empty cells).

Improve the function such that each row of the board is printed in its own line with aligned column entries, zeroes replaced by spaces, and the subgrids visible. For example, the output for the above small and big boards could be:

```
>>> print_board(small)
-------
|  |1 |
|4 |  |
-------
|  | 2|
| 3|  |
-------
>>> print_board(big)
-------------
|   |   |   |
|4  |789|   |
|78 |   | 56|
-------------
| 2 |36 |8  |
|  5|  7| 1 |
|8  |2  |  5|
-------------
|  1|64 |97 |
|   |9  |   |
|   | 3 |  2|
-------------
```

For `k=4`, it is more complicated to have the columns aligned. One could introduce additional spaces for entries less than 10. However, a solution that provides a nicer overview is to use letter codes for the numbers 10 to 16. For example:

```
>>> print_board(giant)
---------------------
| 5  | 4 8| 6  |  9G|
|1   |   D|4  7|F 8 |
|D   | 73 |   9|5A  |
| BCF|A   |  5 |34 D|
---------------------
|F 13|  72|    | 5  |
|   C| 3 5| B E|   9|
|47  |    |C FG|    |
|    |E F |69  |  C |
---------------------
|3 F4| DE |   1|  78|
|    |    |  9A|    |
|B GA|    | 7  | 35 |
|  D |    |E FG| 9 1|
---------------------
|9 2 | E 4|8   |    |
| E  |   A|9 3 |  17|
|8   |G  1|2EB4|   3|
|   1|  5 | G 6| C  |
---------------------
```

Hint: It will be useful to have the built-in `print` function print values without moving the output to a new line. You can do that via `print(x, end='')`, i.e., by using the "optional keyword parameter" `end`. See also the help of the `print` function.

Marking criteria:

- 1 mark for displaying all rows of the field in individual lines
- 1 mark for displaying spaces instead of zeroes and no commas
- 1 mark for visualisation of subgrid boundaries for `k==2`
- 1 mark for visualisation of subgrid boundaries for arbitrary `k` (not necessarily aligned columns for `k > 2`)
- 1 mark if function correctly aligns columns for `k = 4` by replacing values `10` to `16` by letters `'A'` to `'G'` (you are not allowed to change the input board, as this would break the `play` function)

B: Implement the rules (5 marks)

The function `play(board)` provided in the template accepts as input a Sudoku board and allows, in a very rudimentary fashion, to play a game of Sudoku via the console. When you inspect the function (or run it) you will find that it has the following bevaviour:

- When the function is called, it prints the input game board (via the function `print_board`)
- Then it goes into an infinite loop where it first waits for user input (via the built-in input function) and then proceeds as follows:
  - On input `'q'` or `'quit'` the loop ends.
  - On inputting three integers `i j x` (separated by a single whitespace character), it updates the game board by setting the value of field `(i, j)` to `x` and prints the updated board.
  - On input `'n' k d` (the character `'n'` followed by integers `k` and `d`, separated by a single whitespace character), it loads a new game board of specified size `k**2` and difficulty (with some limited available choices for `k` and `d`).

What is unsatisfactory is that the function allows the player to modify the board with invalid moves. Also, the program does not notice when the Sudoku has been solved. In short, it doesn't actually now how Sudoku works. Let's change that.

Write a function `subgrid_values(board, r, c)` that accepts as **input parameters** a Sudoku board `board`, a row index `r`, and a column index `c`, and that produces as **output** a list of all numbers that are present in the subgrid of the board related to the field in row `r` and column `c`. For example:

```
>>> subgrid_values(small, 1, 3)
[1]
>>> subgrid_values(big, 4, 5)
[3, 6, 7, 2]
>>> subgrid_values(giant, 4, 5)
[7, 2, 3, 5, 14, 15]
```

Write a function `options(board, r, c)` that accepts as **input parameters** a game board (board), an integer row index `r`, and an integer column index `c` and that produces as **output** a list of all possible values that a player is allowed to place into field `(r, c)` of the board (think about what that means for an already occupied field). For example:

```
>>> options(small, 0, 0)
[2, 3]
>>> options(big, 6, 8)
[3, 8]
>>> options(giant, 1, 5)
[2, 5, 6, 9, 11, 12, 16]
```

Now modify the play function that on user input of three numbers `i j x`, it only modifies and reprints the board if `x` is a legal value for the field `(i, j)` in the current board and otherwise prints an error message.

Finally, let the play function print a success message when, at the beginning of the loop, the sudoku is solved (i.e., there are no empty fields left).

Marking criteria:

- 2 mark for correct implementation of `subgrid_values` (1 marks if it only works for a specific board size, meaningful explanation of general case in docstring of no more than 200 characters gives second mark even if function implementation is not entirely correct)
- 1 mark for correct implementation of options
- 1 mark for integration into the play function
- 1 mark for success message when board is solved

C: Undo (2 marks)

Now that our program has the essentials of Sudoku internalised, we can add a bit of extra comfort for the player. Presently, it is very inconvenient for players that when they make a mistake they have to reload the board completely (using the `'n' k d` input). Extend the behaviour of the play function such that:

- on input `'r'` or `'restart'` the game restarts with the last loaded game board,
- on input `'u'` or `'undo'` the last move is taken back.

Marking criteria:

- 1 mark for restart functionality
- 1 mark for undo functionality

D: Hints (3 marks)

Even with the added convenience, Sudoku is still hard, especially for beginners. Let us implement some functionality to help the player getting started. Extend the behaviour of the play function such that on input `'h'` or `'hint'` it prints the coordinates of a field for which it is easiest to see the correct value. Think of meaningful ways to implement this functionality and explain your approach in a docstring (either in the play function or, as preferred option, in a function that is called by the function play to receive the hint). For improved convenience, the functionality should also reprint the board with the indicator symbol `'*'` at the position referred to by the hint.

For instance, when entering `'h'`, the function could print the following to the console (this is just an illustration; your function might behave differently):

```
-------------
|   |   |   |
|4  |789|   |
|78 |   | 56|
-------------
| 2 |36 |8  |
|  5| *7| 1 |
|8  |2  |  5|
-------------
|  1|64 |97 |
|   |9  |   |
|   | 3 |  2|
-------------
(4, 4)
```

Marking criteria:

- 2 marks for printing a hint and explaining how the hint is chosen in the docstring (1 mark)
- 1 mark for printing board with marker on the field corresponding to the hint

Part II - Solve and Generate (max 12 marks)

A: Inference (6 marks)

Inference is the process of deriving a logical conclusion from a set of given facts. In the context of Sudoku, a player attempts to infer the correct values for empty fields given the content of the already filled fields. There are numerous inference rules that experienced Sudoku players use (see, e.g., `http://www.taupierbw.be/SudokuCoach/SC_index.shtml` for an overview). Let us focus on the 'Singles' strategy (`http://www.taupierbw.be/SudokuCoach/SC_Singles.shtml`), which contains two separate rules.

The first one, let us refer to it as "forward single", is the simple observation that if an empty field `f` has only one available option `x`, we know that the solution of the board contains `x` in field `f`. Here, options refers to the options available after removing all numbers already present in the row, column, and subgrid of `f`, as implemented in the `options` function in the first part of the assignment. While very intuitive and useful, this rule alone is usually not enough to solve any but the most easy sudokus.

The second single rule, let us refer to it as "backward single", is based on the fact that every region of a Sudoku board, i.e., row, column, and subgrid, the solution *needs to* contain every number from `1` to `n`. From this we can derive the rule as: if within a given region the number `x` is available as option only in one field `f` of that region, then the solution contains `x` in field `f` (because it is the only field that can "supply" `x`). It turns out that both rules in conjunction are enough to solve a lot of Sudoku boards and we can implement a decent partial Sudoku solver based on them.

Start by implementing a function `value_by_single(board, i, j)` that accepts as **input** an `n`×`n`-Sudoku board `board`, a row index `i`, and a column index `j` and that produces as **output** a number from `1` to `n` if that number can be inferred by either of the two single rules to be the solution of the board at field `(i, j)`. If no number can be inferred then the function should return `None`. For example:

```
>>> value_by_single(big, 0, 0)
>>> value_by_single(small, 0, 1)
2
>>> value_by_single(small, 0, 0)
3
```

Next, write a function `inferred(board)` that accepts as input a Sudoku board and returns as output a new Sudoku board that contains all values that can be inferred by repeated application of the two single rules. That is, in the result board `value_by_single` returns `None` for all empty fields. For example, for the big game board above the function would completely solve the board:

```
>>> inferred(big)
[[2, 1, 3, 4, 5, 6, 7, 8, 9],
 [4, 5, 6, 7, 8, 9, 1, 2, 3],
 [7, 8, 9, 1, 2, 3, 4, 5, 6],
 [1, 2, 4, 3, 6, 5, 8, 9, 7],
 [3, 6, 5, 8, 9, 7, 2, 1, 4],
 [8, 9, 7, 2, 1, 4, 3, 6, 5],
 [5, 3, 1, 6, 4, 2, 9, 7, 8],
 [6, 4, 2, 9, 7, 8, 5, 3, 1],
 [9, 7, 8, 5, 3, 1, 6, 4, 2]]
```

In contrast, in the following harder puzzle, our inference rules get stuck right in the beginning:

```
>>> big2 = [[0, 0, 0, 6, 0, 0, 2, 0, 0],
...         [8, 0, 4, 0, 3, 0, 0, 0, 0],
...         [0, 0, 0, 0, 0, 9, 0, 0, 0],
...         [4, 0, 5, 0, 0, 0, 0, 0, 7],
...         [7, 1, 0, 0, 0, 0, 0, 0, 0],
...         [0, 0, 3, 0, 5, 0, 0, 0, 8],
...         [3, 0, 0, 0, 7, 0, 0, 0, 4],
...         [0, 0, 0, 0, 0, 1, 9, 0, 0],
...         [0, 0, 0, 2, 0, 0, 0, 6, 0]]
>>> inferred(big2)
[[0, 0, 0, 6, 0, 0, 2, 0, 0],
 [8, 0, 4, 0, 3, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 9, 0, 0, 0],
 [4, 0, 5, 0, 0, 0, 0, 0, 7],
```

```
[7, 1, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 3, 0, 5, 0, 0, 0, 8],
[3, 0, 0, 0, 7, 0, 0, 0, 4],
[0, 0, 0, 0, 0, 1, 9, 0, 0],
[0, 0, 0, 2, 0, 0, 0, 6, 0]]
```

Finally, integrate this function into the play function such that on user input `'i'` (or `'infer'`), the current board state is replaced by the inferred board state (and as usually the new state is printed to the console to be inspected by the user). For that, make sure that the inferred function does not modify the input board.

Marking criteria:

- 1 mark for correct implementation of the forward single rule
- 3 marks for correct implementation of the backward single rule (1 mark for meaningful structure of algorithm, 1 mark for considering all relevant fields in all related regions, 1 mark for correctly determining the options not provided by another field in the same region)
- 2 marks for the inferred function (1 mark for correct looping, 0.5 mark for correct single iteration) and its integration into the play function (0.5 marks)

B: Backtracking (6 marks)

In cases the like the above, when the implemented logical inference rules are insufficient to solve the board, we have to resort to some degree to search via trial and error. Sudoku is perfectly suited to do this via the backtracking approach presented in the lecture.

Extend the behaviour of the play function such that on input `'s'` or `'solve'` it solves the current board (and prints the solution to be inspected by the user). Make sure to delegate the actual work to solve the board to an appropriately defined function and document how that function works with a short meaningful docstring (at most 300 characters).

Any suitable implementation of backtracking should be enough to solve the small boards. For harder larger boards, smart algorithmic choices can make a big difference. In particular, think about what fields your backtracking algorithm should attempt to fill first. If you have implemented a good selection strategy, give a short explanation in the docstring (max. 250 characters).

For the hardest boards (like `giant3` from the template), you will need to combine backtracking with the logical inference from the previous task. Modify your backtracking function to include inference and give a short explanation in the docstring (max. 300 characters) of why this approach works and how it reduces the computational complexity.

Marking criteria:

- 2 mark for meaningful adoption of backtracking with correct detection of solution state (0.5), meaningul definition of options per call (1 mark), and integration into the play function (0.5 marks). A short (up to 300 characters) explanation in the docstring of the backtracking function that describes the critical choices is worth up to one mark.
- 2 mark for a smart selection of options (by picking the right field to fill) that leads to reduction of computational cost, one mark of which is awarded for a short (up to 250 characters) explanation in the docstring of why the choise is useful.
- 2 mark for cost reduction by integrating inference into backtracking, one of which is granted for a short (up to 300 characters) explanation of why inference can be included in backtracking and how it is useful to reduce the computational complexity.

C: Generate (6 marks)

In this final task we tackle the challenging problem of generating new valid sudokus. For this you will likely want to build on your solution from Task C.

Extend the behaviour of the `play` function such that on input `'g' k`, i.e., the letter `g` followed by an integer `k > 1`, or (`'generate' k`), it generates a new random `k**2`×`k**2` Sudoku board such that

- the board has a unique solution,
- no number of the board can be removed (or set to 0 in terms of our representation) such that the resulting game board would still have a unique solution.

Hints:

- Think of a good decomposition of this problem. Some ideas for useful subproblems are, e.g., generating a random full game board that is valid (i.e., satisfies the placement constraints of Sudoku) and checking whether a given partially filled board has a unique solution.
- For randomising the generation procedure you can use the function `shuffle` from the `random` module. How can you use this within the backtracking solver from the previous task to obtain a random solution?
- You might start with a more or less straightforward approach for checking whether a board has a unique solution. For practical efficiency for `k=3` and especially for `k=4`, think then about how this approach can be improved. The backtracking approach from the lecture can be used to find all solutions to a puzzle. Can you modify this to efficiently detect whether there is more than one solution?
- Your initial solutions to this problem are likely rather slow. Make sure to first test with the smallest board size (`k=2`) and make sure that your code can produce some intermediate console output (via the `print` function) to allow you to diagnose if your computations are still progressing or if they are caught in an infinite loop.

Marking criteria:

- 2 marks for generation of random full Sudoku board
- 2 marks for efficient detection of whether candidate board has unique solution
- 2 marks for overall generation procedure
- For each item half of the marks can be awarded for a meaningful short explanation in the corresponding docstrings