FIT1045 Algorithms and programming in Python, S1-2019
Assignment 1 (value 12%)
Due: Friday 12th Apr 2019, 11:55 pm.

Objectives

The **objectives of this assignment** are:

- To demonstrate the ability to implement algorithms using basic data structures and operations on them.
- To gain experience in designing an algorithm for a given problem description and implementing that algorithm in Python.

Submission Procedure

1. Put you name and student ID on each page of your solution.
2. Save your files into a zip file called yourFirstName_yourLastName.zip
3. Submit your zip file containing your solution to Moodle.
4. Your assignment will not be accepted unless it is a readable zip file.

Important Note: 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.

Marks: This assignment will be marked both by the correctness of your code and by an interview with your lab demonstrator, to assess your understanding. This means that although the quality of your code (commenting, decomposition, good variable names etc.) will not be marked directly, it will help to write clean code so that it is easier for you to understand and explain.
This assignment has a total of 30 marks and contributes to 12% of your final mark. Late submission will have 10% off the total assignment marks per day (including weekends) deducted from your assignment mark. (In the case of Assignment 1, this means that a late assignment will lose 3 marks for each day (including weekends)). Assignments submitted 7 days after the due date will normally not be accepted.
Detailed marking guides can be found at the end of each task. Marks are subtracted when you are unable to explain your code via a code walk-through in the assessment interview. **Readable code is the basis of a convincing code walk-through.**

Task 1: Race to Pi (12 Marks)

Background
The ratio of a circle's circumference to its diameter, usually denoted by the Greek letter π, is an extremely important constant in mathematical computations. Unfortunately, the exact value of π cannot be stored explicitly in a computer, because it is *irrational*, i.e., it does not correspond to any ratio between two integer values. To help us out, there are many different ways to *approximate* π, most of which based on computing a finite prefix of an infinite sum or product. For instance:

1. via a solution to the **basel** problem: $\frac{\pi^2}{6} = \frac{1}{1} + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \ldots$
2. derived from the **taylor** expansion of $1/(1+x)$: $\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$
3. via the **wallis** algorithm: $\frac{\pi}{2} = \left(\frac{2\times2}{1\times3}\right) \times \left(\frac{4\times4}{3\times5}\right) \times \left(\frac{6\times6}{5\times7}\right) \times \left(\frac{8\times8}{7\times9}\right) \times \ldots$
4. via the **spigot** algorithm: $\frac{\pi}{2} = \left(\frac{1}{1}\right) + \left(\frac{1}{1}\times\frac{1}{3}\right) + \left(\frac{1}{1}\times\frac{1}{3}\times\frac{2}{5}\right) + \left(\frac{1}{1}\times\frac{1}{3}\times\frac{2}{5}\times\frac{3}{7}\right) + \ldots$

Each of these sums (or the product in case of 3.) *converges* to the respective value on the left-hand-side of the equals sign. That is for every desired precision $\epsilon > 0$, the absolute difference between the left-hand-side and the sum/product on the right-hand-side is guaranteed to be less than ϵ if a sufficient number of terms is computed. Thus, they are all valid ways to compute π. However, the *speed of convergence*, i.e., the number of terms that have to be computed to reach a desired precision is different for each method. (Note that in case of *wallis* and *spigot* every expression in parenthesis is considered one term; e.g., the first term of the wallis product equals $4/3$, the third term of the spigot sum equals $2/15$.) In this task, we will determine which of the above methods is preferable from this perspective.

Instructions
Create a Python module called `pi.py`. Within that module:

a) Write four different functions for approximating π corresponding to the four different approaches listed above (the functions must be called `basel`, `taylor`, `wallis`, and `spigot`). Each of these functions must obey the following input/output specification:

Input: a float parameter `precision` which specifies the precision to which π should be calculated, i.e., the absolute difference of the approximation to the Python constant `pi`.

Output: two values `(x,n)` such that `x` is the first approximation to π with an absolute difference from `pi` of at most `precision`, and `n` is the number of terms of the sum/product that were computed in the process.

For example, if the input was 0.1, then the function should continue to compute terms of the series until the approximation `x` to π is less than 0.1 away from the Python constant `pi` (see more detailed examples below).

b) Write another function called `race(precision, algorithms)` that can be used to compare the convergence speed of different approximation algorithms via the following input/output specification:

Input: a float parameter `precision` and a list of Python functions `algorithms`.

Output: a list of integer pairs `[(i1, n1), (i2, n2),..., (ik,nk)]` such that

- for each pair `(i,n)` in the list, n is the number of steps that the `i`-th algorithm takes to approximate π within the given precision
- the list is in *ascending order* with respect to the number of steps (the second number of the pair) with ties resolved arbitrarily.

Note that the input to `race` are functions and not strings. That is, in principle `race` can be used for evaluating arbitrary functions for approximating π, not only the four covered here.

c) Write a function `print_results` which takes as input the output of the function `race` and prints the results in a human readable format. Each line of the output should be "`Algorithm i finished in n steps`" where `i` is first element of the integer pair, and `n` is the second.

The only two import statements allowed in your module are `from math import pi` and `from math import sqrt`. Note that the subtasks can be solved independently and the given order (a, b, c) is only a recommendation.

Examples

a) Calling `wallis(0.2)` returns `(2.972154195011337, 4)` as the result of approximating π with

$$2.972154195011337 = 2\underbrace{\left(\frac{2\times 2}{1\times 3}\times\frac{4\times 4}{3\times 5}\times\frac{6\times 6}{5\times 7}\times\frac{8\times 8}{7\times 9}\right)}_{4\text{ terms}}$$

using 4 terms because `pi - 2.972154195011337 < 0.2` and non of the three prior wallis approximations satisfies this condition.

b) Calling `basel(0.1)` returns `(3.04936163598207, 10)` as the result of approximating π with

$$3.04936163598207 = \sqrt{6\underbrace{\left(\frac{1}{1}+\frac{1}{4}+\frac{1}{9}+\frac{1}{16}+\cdots+\frac{1}{100}\right)}_{10\text{ terms}}}$$

using 10 terms because `pi - 3.04936163598207 < 0.1` and non of the nine prior basel approximations satisfies this condition.

c) Calling `taylor(0.2)` returns `(3.3396825396825403, 5)` as the result of approximating π with

$$3.3396825396825403 = 4\underbrace{\left(\frac{1}{1}-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}\right)}_{5\text{ terms}}$$

using 5 terms because `pi - 3.3396825396825403 < 0.2` and non of the four prior taylor approximations satisfies this condition.

d) Calling `spigot(0.1)` returns `(3.0476190476190474, 4)` as the result of approximating π with

$$3.0476190476190474 = 2\underbrace{\left(\frac{1}{1}+\left(\frac{1}{1}\times\frac{1}{3}\right)+\left(\frac{1}{1}\times\frac{1}{3}\times\frac{2}{5}\right)+\left(\frac{1}{1}\times\frac{1}{3}\times\frac{2}{5}\times\frac{3}{7}\right)\right)}_{4\text{ terms}}$$

using 4 terms because `pi - 3.0476190476190474 < 0.1` and non of the prior three spigot approximations satisfies this condition.

e) Calling `race(0.01, [taylor, wallis, basel])` returns `[(2,78),(3,96),(1,100)]` because functions `taylor`, `basel`, and `wallis` need 100, 96, and 78 terms, respectively, to approximate π within the required precision of 0.01.

f) Calling `print_results([(2,78),(3,96),(1,100)])` prints

```
Algorithm 2 finished in 78 steps
Algorithm 3 finished in 96 steps
Algorithm 1 finished in 100 steps
```

Marking Guide (total 12 marks)

Marks are given for the correct behaviour of the different functions:

(a) 2 marks each for `basel`, `taylor`, `wallis`, and `spigot`

(b) 3 marks for `race`

(c) 1 mark for `print_results`

All functions are assessed independently. For instance, even if function `race` does not always produce the correct output, function `print_results` can still be marked as correct.

Task 2: Reversi (18 Marks)

Background
Reversi is a game for two players (Black and White), which is played on a squared board of 8 rows/columns with stones that are white on one side and black on the other. In the beginning of the game, stones of Player 1 (B) and Player 2 (W) are placed in a cross arrangement at the center of the board as shown in Table **??**. Then the players take turns placing stones on empty fields on the board with their colour facing up according to the following rules:

1. All opponent's stones that are enclosed in a straight line (horizontal, vertical, or diagonal) between the newly placed stone and another stone of the same color are flipped (see Table **??**).
2. A move is only valid if it turns at least one opponent's stone.
3. If a player has no valid moves, their turn is skipped (see example in Table **??**).
4. The game ends when both players have no legal moves.

When the game ends, each player scores points equal to the number of stones which have their colour face up, and the player with a higher score wins (draw if scores equal). In this task you will create a Python program that allows one or two players to play a game of Reversi.

	a	b	c	d	e	f	g	h
1								
2								
3								
4				W	B			
5				B	W			
6								
7								
8								

Table 1: Initial board state at the beginning of game. Player 1 (B) has the first move.

	a	b	c	d	e	f	g	h
1								
2								
3								
4				W	B			
5				B	B			
6					B			
7								
8								

Table 2: Board state after Player 1 (B) played "e6" as her first move.

	a	b	c	d	e	f	g	h
1								
2								
3								
4				W	B	•		
5				B	B			
6				•	B	•		
7								
8								

Table 3: Valid moves (indicated by •) for Player 2 (W) following "e6" from Player 1.

	a	b	c	d	e	f	g	h
1								
2						W	W	
3	W	W	W	W	W	W	W	B
4			W	W	W	W		B
5			W	W	W			B
6								
7								
8								

Table 4: Board state with no valid move for Player 2 (W), who, in case it is her turn, has to pass back to Player 1 (B), whose only valid move is "e1".

Instructions
Create a python module `reversi.py`. The only import statement allowed in your module is `import copy`. Your module must at least contain the nine functions described in the subtasks below, but you are allowed, and in fact, encouraged, to implement additional functions as you deem appropriate.

a) Create three functions `new_board()`, `print_board(board)`, and `score(board)` that represent the basic elements of Reversi as follows. The function `new_board()` takes no parameters and returns a fresh board configuration representing the Reversi starting position. The other two functions each take as input a board configuration. The function `score(board)` returns as output a pair of integers `(s1, s2)` where `s1` represents the number of stones of Player 1 in the given board configuration and `s2` represents the number of stones of Player 2. The function `print_board(board)` has no return value but prints the given board configuration in human-readable form to the console (in particular, all rows of the board must be printed in separate lines of the console and labelled by their names 1, 2, ... and all columns must be properly aligned and labelled by their names a, b, ... — similar to the illustrations above). For these and all other functions of this task, a **board configuration** must be represented as a list of lists in the following way:

- A board configuration is a list of length 8, the elements of which represent the rows of the board from top to bottom: the first element represents the topmost row of the board, row 1, the second represents row 2, etc..
- Each row, i.e., each element of a list representing a board configuration, is also a list of length 8. These inner lists represent the rows of the board from left to right: the first element represents the field in the leftmost column, Column A, the second element represents the field in second column, Column B, etc.
- Each field, i.e., each element of the inner lists representing the rows, is an integer from the set $\{0, 1, 2\}$ representing the state of that field: 0 represents an empty square, 1 represents a square with a Black stone, and 2 represents a square with a White stone.

See example **??** for an illustration of this representation of a board configuration.

b) Create three functions `enclosing(board, player, pos, direct)`, `valid_moves(board, player)`, and `next_state(board, player, pos)` that represent the rules of Reversi according to the following specifications.

- A board **position** is represented by a pair `(r, c)` where `r` and `c` are each integers from the range `range(8)` representing a row index and a column index, respectively. For instance, the position "b3" is represented by `(2, 1)`. The row and column order of the "b3" style and the specification of `position` is reversed. This is because it is natural to constuct lists of lists in such a way that the outer lists correspond to rows, and the inner list positions corerspond to column positions, so we reference them with row first, then column. This row, column convention is quite common across maths and computer science.
- A **direction** is represented by a pair `(dr, dc)` where `dr` and `dc` are each integers from the set {`-1, 0, 1`}. For instance, "horizontal to the left" is described by `(0, -1)`, "diagonal to the right and down" is described by `(1, 1)`, and so on).
- The function `enclosing(board, player, pos, dir)` represents whether putting a player's stone on a given position would enclose a straight line of opponent's stones in a given direction:

  Input: A board configuration `board`, an integer `player` from the set {`1, 2`}, a board position `pos` and a direction `dir`.

  Output: `True` if `board` contains a stone of `player` in direction `dir` from position `pos` and all positions on the straight line between that stone and `pos` contain stones of the other player, and there is at least one stone belonging to the other player on the straight line; `False` otherwise.
- The function `valid_moves(board, player)` has:

  Input: A board configuration `board` as well as an integer `player` from the set {`1, 2`}.

  Output: A list of board positions (represented as pairs of integers as described above) `[(r1, c1), (r2, c2), ..., (rk, ck)]` such that the elements of that list correspond to all valid positions that Player `player` is allowed to drop a stone on in the given input configuration `board` (in particular, the list is empty if the player has not valid moves in the given board).
- The function `next_state(board, player, pos)` has:

  Input: A board configuration `board` as well as an integer `player` from the set {`1, 2`} and a board position representing the move of Player `player` to place one of their stones on the board configuration `board` on the field described by `pos`.

  Output: A pair `(next_board, next_player)` such that `next_board` is the board configuration resulting from Player `player` placing a stone on position `pos` on the input board configuration `board`, and `next_player` is an integer from the set {`0, 1, 2`} corresponding to the player who gets to move next or 0 in case the game ends after the input move is carried out.

c) Create functions `position(string)`, `run_two_players()` and `run_single_player()` that put all of the above together and that allow one or two players to play a game of Reversi through the console as follows.

- The function `position(string)` accepts as input any string `string` (e.g., `"b5"` or `"Genghis Khan"`) and has as output the board position `(r, c)` described by the string or `None` in case the string does not correspond to a valid board position.
- Running `run_two_players()` should repeatedly do the following until the user selects to stop:
  1: Print the current board state
  2: Ask the player whose turn it is to input a position to drop a stone (in the format `"d4"`). The player can also choose to quit by entering `"q"`
  3: Your program should then respond in one of the following ways:
  – If the move is not valid, print `"invalid move"`
  – If the move is valid, place the move on the board.
  4: If the game ended (no valid move remaining for either player), print the game result and exit
  5: If the game did not end, the current player's turn is over. It is now the other player's turn
- Running `run_single_player()` should behave identical to `run_two_players()` except that all moves of Player 2 are carried out automatically such that the resulting board configuration has the *maximal score* for Player 2 (among all their valid moves).

Examples

a) Calling `score(new_board())` returns `(2, 2)` because the score in the starting configuration is 2 for Player 1 (B) and 2 for player 2 (W).

b) Calling `enclosing(new_board(), 1, (4, 5), (0, -1))` returns `True` because Player 1 can enclose one opponent's stone to the left when placing a stone on field "f5". In contrast, there is no straight line of opponent's stones to enclose on the diagonal line to the bottom right. Therefore, `enclosing(new_board(), 1, (4, 5), (1, 1))` returns `False`.

c) Calling `next_state(new_board(), 1, (4, 5))` will return `(next_board, 2)` where `next_board` is the board configuration

```
[[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0],
[0,0,0,2,1,0,0,0],
[0,0,0,1,1,1,0,0],
[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0]]
```

because that is the configuration resulting from Player 1 putting a stone on field "f5" (encoded by integer pair `(4, 5)`) in the starting configuration, and Player 2 has the next move afterwards.

d) Calling `valid_moves(next_state(new_board(), 1, (4, 5))[0], 2)` returns the list `[(3, 5), (5, 3), (5, 5)]` or any of its permutations (list with same elements but different order) because the valid moves of Player 2 after Player 1 played "f5" in Turn 1 are "d6", "f4", and "f6".

e) Calling `position("e3")` returns the pair `(2, 4)` because that is the internal representation of board position "e3". Calling `position("l1")`, `position("a0")`, or `position("Genghis Khan")` all return `None` because all these input do not correspond to valid board positions.

Marking Guide (total 18 marks)
Marks are given for the correct behaviour of the different functions:

a) 1 mark each for `new_board`, `print_board`, and `score`

b) 4 marks for `enclosing`, 3 marks for `valid_moves`, and 2 marks `next_state`

c) 1 mark for `position`, 2 marks for `run_two_players` and 3 marks for `run_single_player`

All functions are assessed independently to the degree possible. For instance, even if function `valid_moves` does not always produce the correct output, function `run_two_players` can still be marked as correct.