CMPUT 175 (LEC B1 B2 B3 B4 Winter 2022)
Participants
Grades
General
Lectures Video streaming with Zoom
Labs
Assignments
Week 1 (January 5-7)- Introduction and back to programming (Chapter 1)
Week 2 (January 10 - 14) - Hands-on Python with fun simple programs
Week 3 (January 17 - 21) - Algorithm Analysis
Week 4 (January 24 - 28) - Data Structures and Classes
Week 5 (January 31 - February 4 - Exception Handling

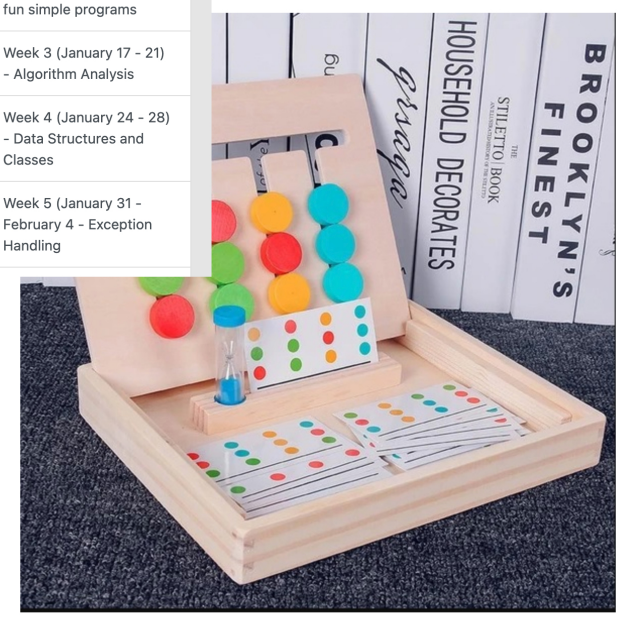
EC B1 B2 B3 B4 - Winter 2022

CMPUT 175 (LEC B1 B2 B3 B4 Winter 2022) / Assignments / Assignment 3

- AbacoStack game

23:55
%
ts allowed

r of Python and hands-on experience with algorithm coding, input validation, exceptions, Stack, Queue, and data structures with encapsulation.



AbacoStack is a game invented for this course so don't bother looking for it online.

You have a structure formed by three side by side stacks that are limited in size. For this game we will limit our stacks to three elements each. For the explanation we will initially limit ourselves to three stacks with a depth of three. You can see above in the picture a physical example with 4 stacks of depth 3.

Back to our explanation, above these three stacks, you have a list of five positions from 0 to 4. The first stack is aligned with position 1; stack 2 with position 2, and stack 3 with position 3. Positions 0 and 4 don't have a stack aligned with them.

See this figure below to have a better idea about how this structure looks.

```
0 1 2 3 4
. . . . .
  . . .
  . . .
  . . .
```

The structure is filled with nine beads: 3 As, 3 Bs and 3 Cs, representing 3 colours. Initially the structure looks like this with the 3 stacks each filled with either all As, Bs, or Cs.

```
0 1 2 3 4
. . . . .
  A B C
  A B C
  A B C
```

The structure always has 9 beads. The beads can move one position at a time to an empty position. An empty position is represented by a "." here.

For example from the initial configuration, one can pop a bead from the first stack and switch it with the top bead in the second stack in the follow series of moves:

```
0 1 2 3 4
. . . . .
  A B C
  A B C
  A B C

0 1 2 3 4
. A . . .
  . B C
  A B C
  A B C

0 1 2 3 4
A . . . .
  . B C
```

```
  A B C
  A B C

0 1 2 3 4
A . B . .
  . . C
  A B C
  A B C

0 1 2 3 4
A B . . .
  . . C
  A B C
  A B C


0 1 2 3 4
A . . . .
  B . C
  A B C
  A B C


0 1 2 3 4
. A . . .
  B . C
  A B C
  A B C

0 1 2 3 4
. . A . .
  B . C
  A B C
  A B C

0 1 2 3 4
. . . . .
  B A C
  A B C
  A B C
```

The game consists of getting a random configuration card and doing the minimum numbers of moves to change the beads of the AbacoStack to that given configuration.

For example, you get the following card:

```
 |A A C|
 |B C A|
 |C B B|
```

You need to move the beads one by one to get

```
from               to
0 1 2 3 4       0 1 2 3 4
. . . . .       . . . . .
  A B C           A A C
  A B C           B C A
  A B C           C B B
```

The game is played by a user, not the computer. The computer would generate a random configuration card and the user would have to solve the game by indicating the moves to do in order to change the configuration from the initial one to the one indicated on the card.

The moves are indicated by the following input pair **ij** where **i** is a digit and **j** a character like the following:

- **1u** means stack 1 upward move
- **1d** means stack 1 downward move
- **2u** means stack 2 upward move
- **2d** means stack 2 downward move
- **3u** means stack 3 upward move
- **3d** means stack 3 downward move
- **0r** means position 0 right move
- **1r** and **1l** mean position 1 right move and left move respectively
- **2r** and **2l** mean position 2 right move and left move respectively
- **3r** and **3l** mean position 3 right move and left move respectively
- **4l** means position 4 left move

Task 1 : The Card Class

1. Implement a class named **Card** representing a configuration card. An instance of this class should have one private property, **beads** a list storing in order the beads in each stack, starting with the first stack to the last, each stack stored top to bottom.
2. The constructor will receive the number of colours and the depth of the stacks. Remember in our game we have the size of the stack set at 3 and we have 3 colours: A, B, and C representing each stack. But you should have a generic solution. The picture above for example shows the game with 4 stacks.

   The constructor initializes a list the size of number of stacks (i.e. colours) times the depth of the stack with the beads then shuffles them. Again for our game it is 3x3=9 but these should not be hard coded. Later the size will be selected by the player.

   The card class should have three methods: ***reset()*** to reshuffle the card to generate a new configuration, ***show()*** to display a card, and ***stack(number)*** to return the ordered list of elements top to bottom in the *number* stack. The card should be displayed as follows:

   ```
   |A A C|
   |B C A|
   |C B B|
   ```

   from a list ['A', 'B', 'C', 'A', 'C', 'B', 'C', 'A', 'B']

   From the list above, *stack(1)* would return: ['A', 'B', 'C'] while *stack(2)* would return ['A', 'C', 'B']. Note that *stack(4)* wouldn't make sense in this case.
3. Implement the __str__() method to convert a Card instance into a string such that the string represents the stacks in the configuration card. For the card above the string should look like: |ABC||ACB||CAB|
4. Write a method *replace*(filename, n) that reads from the file filename the n^{th} config card and replaces the card state with the new configuration. Each line in the file has one card defined as "C C C C C C C C C" where C is a coloured bead A, B, or C. Like an index, n starts with 0. Make sure your input is correct when you read it from the file.
5. Test your class in isolation. Be sure to leave your test code with your class submission.

Task 2: The Bounded Stack Class

Task 2: The Bounded Stack Class

1. Implement a class named **BStack**. An instance of this class is a stack with a limited capacity. Like the bounded queue you saw in class, this bounded stack will receive a maximum capacity as a parameter, store its capacity limit and check this whenever we push elements. Also don't forget to implement ***isFull()***.
2. Test your class in isolation. Be sure to leave your test code with your class submission.

Task 3: The AbacoStack Class

1. Implement a class named **AbacoStack**to represent the AbacoStack structure. An instance of this class should store bounded stacks and a list representing the top row. The size of the list will be the number of bounded stacks + 2. The number of bounded stacks and their common depth should be given as input to the constructor of this class. The structure should be initialized so that each stack has only one and unique colour (colours are represented by letters A, B, C, etc.). Another property of an instance of this class is to store the numbers of moves already done since the initialization. Initially, it should be 0.
2. Implement a method *moveBead(move)* which changes the state of the AbacoStack instance based on the valid moves indicated above. *move* is a string of two characters. Raise an Exception "Error: invalid move" if the move cannot be done. For example **2d** cannot be done if Stack2 is full, or **1r** cannot be done if there is a bead in position 2 of the top list.
3. Implement a method *isSolved(card)* which returns TRUE if the state of the instance corresponds to the configuration card, FALSE otherwise. *card*is an instance of the Card class.
4. Implement the *reset()* method which resets the property *moves* to zero and rearrange the stack to the initial position with each stack having its own beads.
5. Implement a method named *show(card)*, that takes an optional parameter *card* and displays the state of the AbacoStack instance. *card* is an instance of the class Card. When the parameter *card* is present, the configuration card will also be displayed on the side of the AbacoStack instance in addition to the number of moves already taken since the start (like below) otherwise only the state of the abacoStack is shown as below.

```
0 1 2 3 4
A . . C .      card
| B . . |    |A A C|
| A B C |    |A B C|
| A B C |    |A B C|
+-------+            34 moves
```

and without the card

```
0 1 2 3 4
A . . C .
| B . . |
| A B C |
| A B C |
+-------+
```

6. Test your class in isolation. Be sure to leave your test code with your class submission.

Task 4: The Game

Write the python program to play the game. To start the game a player has to choose the size of the AbacoStack by selecting the number of stacks between 2 and 5 and selecting the depth of the stacks between 2 and 4.

Here are all the possibilities. Notice that some can be very difficult if not impossible to solve. This why we recommend the 3 by 3 or 4 by 3 configurations.

```
0 1 2 3    0 1 2 3 4   0 1 2 3 4 5    0 1 2 3 4 5 6
. . . .    . . . . .   . . . . . .    . . . . . . .
| A B |    | A B C |   | A B C D |    | A B C D E |
| A B |    | A B C |   | A B C D |    | A B C D E |
+-----+    +-------+   +---------+    +-----------+


0 1 2 3    0 1 2 3 4   0 1 2 3 4 5    0 1 2 3 4 5 6
. . . .    . . . . .   . . . . . .    . . . . . . .
| A B |    | A B C |   | A B C D |    | A B C D E |
| A B |    | A B C |   | A B C D |    | A B C D E |
| A B |    | A B C |   | A B C D |    | A B C D E |
+-----+    +-------+   +---------+    +-----------+


0 1 2 3    0 1 2 3 4   0 1 2 3 4 5    0 1 2 3 4 5 6
. . . .    . . . . .   . . . . . .    . . . . . . .
| A B |    | A B C |   | A B C D |    | A B C D E |
| A B |    | A B C |   | A B C D |    | A B C D E |
| A B |    | A B C |   | A B C D |    | A B C D E |
| A B |    | A B C |   | A B C D |    | A B C D E |
+-----+    +-------+   +---------+    +-----------+
```

Once the size of the AbacoStack is selected, the program should iteratively generate a new random configuration card and allow the user to solve the puzzle.

The user should be given the option to either play one move, a sequence of moves (max 5 at a time), reset the state game to retry the same card, or quit. For example, the user could be prompted:

Enter your move(s) [Q for quit and R to reset]:

Q would indicate quitting, R would indicate resetting and one or more moves separated by a space would provide the move or moves to do. If more than 5 moves are provided, only the first 5 are considered and the others are ignored.

The moves are given as specified above. If a move is not valid, the state of the of the game should not change. If a sequence of moves is given, the state of the game should change to the new state after all those moves are done if all moves are valid in that sequence, otherwise the moves are interrupted at the first illegal move. We would then display the state up to the last legal move that was read.

After each move or the sequence of moves is played, the state of the game should be displayed with the number of moves done since the start. If a player enters a sequence of moves, all the sequence of legal moves should be played even if the winning position is reached before the last move in the sequence. Therefore, testing whether the puzzel is solved should be done after the sequence of moves.

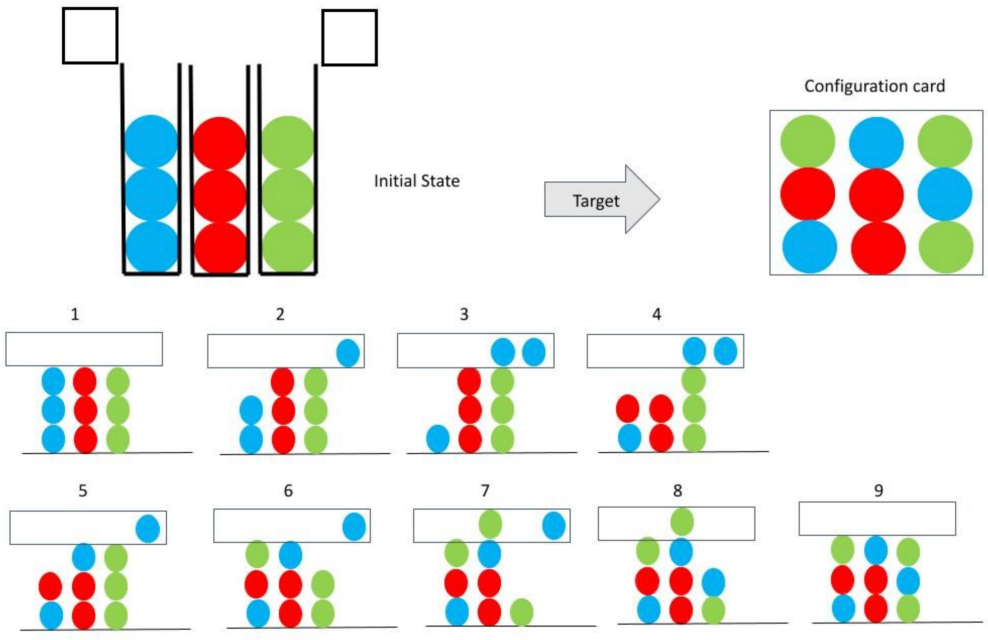
Once the puzzle is solved, after one single move or a sequence of moves, the program should congratulate the player, show the AbacoStack finished with the number of moves played, and ask whether the user would like to get another configuration card to attempt, to which the user should reply by a Y or N for yes or no.

Task 5: Optional

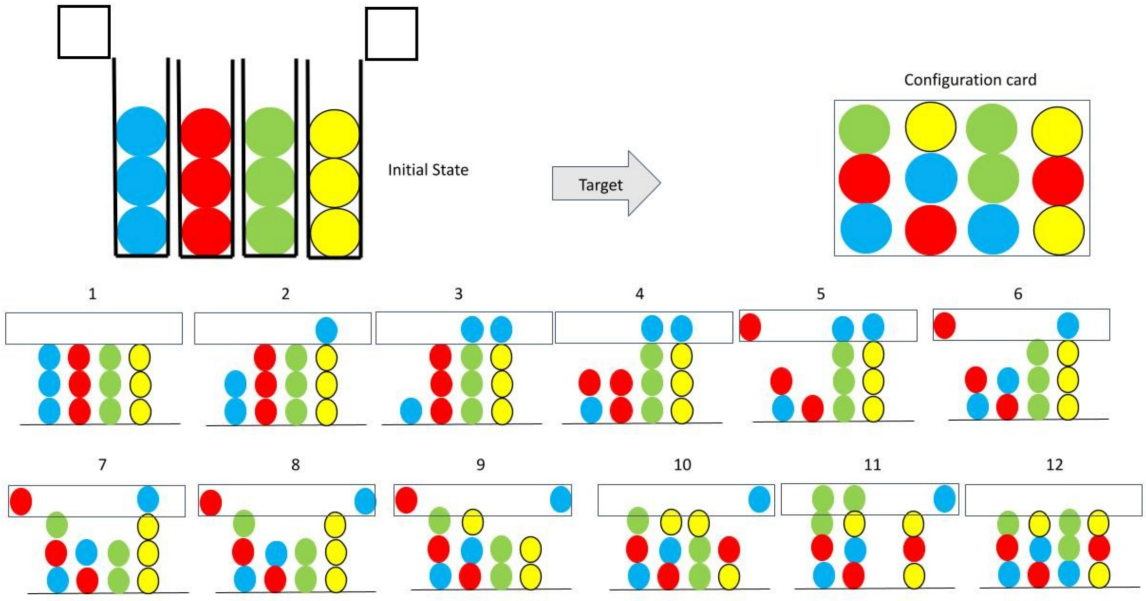
You can add the number of minutes and seconds to solve the problem by getting the current time from the system when you start the game and display the card the first time. Then the spent time is the new current time minus the start time. Investigate online how you could do that. There are also hints on the slides on algorithm analysis.

Examples added:

Solving an AbacoStack of 3 stacks with depth 3.



Solving an AbacoStack of 4 stacks with depth 3.



Questions raised in the forum about the Assignment

1-

General Guidelines

In addition to making sure that your code runs properly, we will also check that you follow good programming practices. For example, divide the problem into smaller sub-problems, and write functions to solve those sub-problems so that each function has a single purpose; use the most appropriate data structures for your algorithm; use concise but descriptive variable names; define constants instead of hardcoding literal values throughout your code; include meaningful comments to document your code, as well as docstrings for all functions; and be sure to acknowledge any collaborators/references in a header comment at the top of your Python file.

You may import the Circular Queue class, if required. To do so, create those classes (as implemented in the lecture notes) in a file called **lectureStructures.py**. Do not submit this file - we will test your code using our own lectureStructures.py, so be sure that the code you use matches the ADT covered in the lecture.

Restrictions for this assignment: Do **NOT** change the method signatures for each class - implement your classes exactly as described in the ADTs. Do not use break/continue. You should import the random module.

Assignment Deliverables

- You will submit two files: **AbacoStack.py** which contains all classes and functions created in Tasks 1-3, and **assignment3.py** which contains the game created in Task 4.
- Note that late submissions *will not be accepted.* You are able to submit your work as often as you like, and only the last submission will be marked. So submit early, and submit often.

Submission status

Attempt number	This is attempt 1.
Submission status	No attempt
Grading status	Not graded
Due date	Friday, 1 April 2022, 11:55 PM
Time remaining	8 days 9 hours
Last modified	-
Submission comments	▸ Comments (0)

Add submission

You have not made a submission yet.