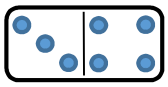
Assignment 2 – Dominoes 175 style

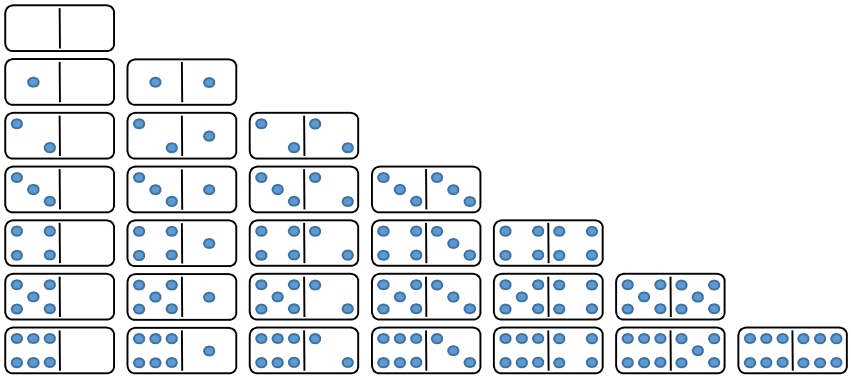
Due Date: Wednesday, March 24th 2021 at 23:55 (Edmonton time)
Percentage overall grade: 7%
Penalties: No late assignments allowed
Maximum marks: 100

Assignment Specification

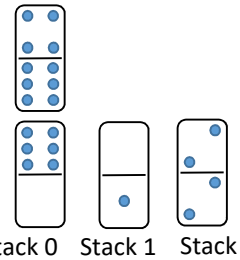
You are tasked with creating a one-player tile-matching game, played with one standard "double-six" set of dominoes. A domino is a tile with two ends, where each end has 0 to 6 dots on it. For example, a domino with 3 dots on one end and 4 dots on the other end is shown below:



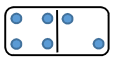
A standard double-six domino set contains the 28 unique dominoes illustrated below:



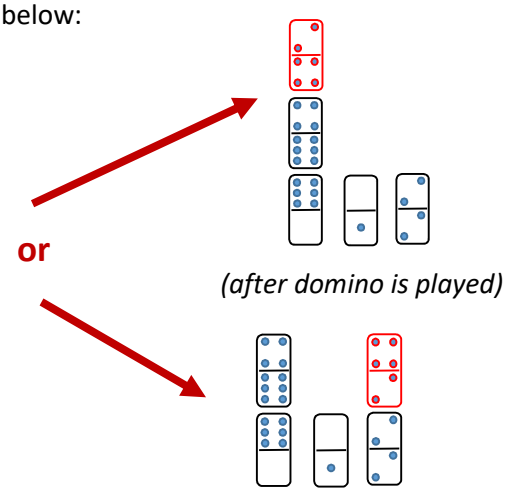
The game will start with all 28 dominoes face down in a grid with 7 columns and 4 rows. (i.e. by face down, we mean that the player can only see the back of the domino, not the side with all of the dots.) The player will choose a domino from the grid and turn it over. The player will then attempt to play the domino by adding it to the top of one of three stacks of dominoes in the play area. If a stack is empty (i.e. it does not currently contain any dominoes), any domino can be added to it and the player can decide on the orientation of the domino (i.e. which end is at the top and which end is at the bottom). If a stack has at least one domino currently in it, a new domino can only be added to the top of the stack if one of its ends has the same number of dots as the top end of the domino that is currently on the top of the domino stack. In that case, the domino must be added to the stack so that the end of the domino with the matching number of dots is oriented to the bottom (i.e. touching the side with the matching number of dots on the domino already on the stack), as illustrated below:



(before domino is played)



This domino can be added to the top of Stack 0 or Stack 2, but not Stack 1.



(after domino is played)

The player continues to draw dominoes from the grid and adding them to a stack in the play area until one of the stacks reaches a height of 6 dominoes (win) or the player cannot play the domino on any of the 3 stacks in the play area (lose).

Task 1: Domino class

Create a Python file called *domino175.py*. Inside this file, create and test a Domino class according to the description below. Note that the following methods form the public interface for this class – you must complete them all as specified, and cannot add additional methods to the public interface. However, you can include additional private helper methods that are only called within this class, if you wish.

`Domino(dotsA, dotsB)` – creates a domino, where the number of dots on each end of the domino are described by the integers `dotsA` and `dotsB`. After asserting that the input parameters are valid, `dotsA` and `dotsB` should be used to initialize two private attributes: `top` and `bottom`. Both of these attributes are integers, and the top is the end with the most dots on it. A third private attribute, `faceDown`, should also be created which indicates if the tile is facing down (True) or facing up (False). A domino should be facing up when it is first created.

`setTop(dots)` – updates which end of the Domino is at the top, according to the `dots` parameter. The integer `dots` must match one of the existing ends of the Domino instance (otherwise an AssertionError is raised). Nothing is returned.

`turnOver()` – updates the Domino instance so that if it was facing up, it will now be facing down. Similarly, if it was facing down, it will now be facing up. Nothing is returned.

`getTop()` – returns the integer number of dots on the top end of the Domino instance.

`getBottom()` – returns the integer number of dots on the bottom end of the Domino instance.

`isFaceDown()` – returns the Boolean value indicating whether the Domino instance is facing down (True) or up (False).

`__str__()` – returns the string representation of the Domino instance. The format of this string should be '[*bottom number of dots*|*top number of dots*]' if it is facing up. For example, a domino with 3 dots on the bottom and 4 dots on top will return the string `'[3|4]'`. Any domino that is facing down will have question marks, '?', instead of the numbers: `'[?|?]'`

Test your Domino class thoroughly before moving on. You may wish to use assertions to verify expected behaviour, like in Lab 7 (Linked Lists), though you don't have to. Place these tests under `if __name__ == "__main__":` in *domino175.py* for the marker to see.

Task 2: DominoDeck class

Create and test a DominoDeck class in *domino175.py*, according to the description below. Note that the following methods form the public interface for this class – you must complete them all as specified, and

cannot add additional methods to the public interface. However, you can include additional private helper methods that are only called within this class, if you wish.

`DominoDeck()` – creates an empty deck of dominoes, capable of holding the 28 dominoes in a standard "double-six" set. Notice that this deck acts very much like a queue, where we deal the front domino and add new dominoes to the rear of the deck. In the `__init__` method, create a single private attribute that is the most time-efficient queue with a maximum capacity that we covered in the lectures. You should import the queues.py file provided with Lab 6 to accomplish this. (You do not need to submit the queues.py file since your marker will also have access to that file.)

`populate(useFile)` – modifies the deck by adding new dominoes face up to the rear of the deck. Nothing is returned.

- If `useFile` is True, the user should be prompted to provide the name of an input text file. If any problem occurs when opening the file, an error message should be displayed and the user should be re-prompted to provide the name of another input text file, until the file can be opened successfully. (See **output_testWin.txt**.) The input text file should contain information about the dominoes that will be added to the deck, in the same order as indicated in the file. (i.e. The first line of the text file corresponds to the first domino that should be added to the deck.) Each valid line of the text file will contain an integer, followed by a forward slash ('/'), followed by another integer, where the integers represent the number of dots on each end of the domino. (See the sample **testWin.txt** provided.) You cannot assume that all lines in the file will be valid, and you cannot assume that there will be information for exactly 28 dominoes in the file. If there is invalid data in the file or not 28 dominoes, you should raise an Exception with the argument "Cannot populate deck: invalid data in xx" (where xx is replaced with the filename), ensure the deck is empty, and close the file before leaving this method. If there is information about exactly 28 valid dominoes in the file, you can assume that those dominoes will form a full "double-six" set which should be added to the deck, in the same order that they appear in the file.

- If `useFile` is False, a complete "double-six" set of dominoes should be created, shuffled, and used to populate the deck.

- If `useFile` is not True or False, an AssertionError should be raised with the argument "Cannot populate deck: invalid argument provided." Nothing should be added to the deck in this case.

`deal()` – modifies the deck by removing the domino from the front of the deck, and returns that front domino, face down. Raise an Exception if the deck is empty, with the argument 'Cannot deal domino from empty deck'. This method should have an O(1) time efficiency.

`isEmpty()` – returns a Boolean value indicating whether there are no dominoes in the deck (True) or if there is at least one domino in the deck (False).

`size()` – returns the integer number of dominoes currently in the deck.

`__str__()` – returns the string representation of all of the dominoes in the deck. You can decide how this string should be formatted, but make it clear which is the front domino and be sure to show the number of dots on each domino (i.e. do not just show all dominoes as face down)

Test your DominoDeck class thoroughly before moving on. Again, place these tests under `if __name__ == "__main__":` in *domino175.py* for the marker to see.

Task 3: DominoStack class

Create and test a DominoStack class in *domino175.py*, according to the description below. Note that the following methods form the public interface for this class – you must complete them all as specified, and cannot add additional methods to the public interface. However, you can include additional private helper methods that are only called within this class, if you wish.

`DominoStack()` – creates an empty stack that will ultimately hold Domino instances. Nothing is returned. Note that you may use inheritance (in which case you should include the code for the Stack class that you are inheriting from in the *domino175.py* file), or you can implement from scratch – either approach is equally acceptable for this assignment.

`peek()` – returns the number of dots on the top of the Domino that is at the top of the stack. Raises an Exception with the argument "Error: cannot peek into an empty stack" if the stack does not contain any dominoes.

`isEmpty()` – returns a Boolean value indicating whether there are no dominoes on the stack (True) or if there is at least one domino on the stack (False).

`size()` – returns the integer number of dominoes currently on the stack.

`push(domino)` – adds the provided `domino` onto the top of the DominoStack if the DominoStack is empty, or if the number of dots on the top of the Domino currently on the top of the stack matches the number of dots on one side of the provided `domino`. If the last case, add the provided `domino` so that its bottom matches the top of the domino below it. Raise an AssertionError with the argument "Can only push Dominoes onto the DominoStack" if the provided `domino` is not an instance of the Domino class. If the `domino` cannot be added to the top of the stack (i.e. the stack has a domino on its top whose top dots do not match the dots on either side of the provided `domino`), raise an Exception with the argument "Cannot play xx on stack", where xx is the string representation of the domino, face up. This method should have an O(1) time efficiency.

`__str__()` – returns the string representation of the DominoStack instance. Specifically, it should return the string representations of any Dominoes on the stack, separated by a single dash (`'-'`). The bottom of the stack should be at the beginning of the returned string, the top should be at the end.

Test your DominoStack class thoroughly before moving on. Again, place these tests under `if __name__ == "__main__":` in *domino175.py* for the marker to see. You now have a domino175 module that you can use for many different domino game programs!

Task 4: Table class

Create a Python file called *assignment2.py*. Inside this file, create and test a Table class according to the description below. Note that the following methods form the public interface for this class – you must complete them all as specified, and cannot add additional methods to the public interface. However, you can include additional private helper methods that are only called within this class, if you wish.

`Table()` – creates a new empty domino game table. This table should have one deck (a DominoDeck) that is originally empty. This table will also have an empty grid, capable of holding 4 rows with 7 columns of dominoes. Finally, this table will have a playing area that consists of 3 DominoStacks, which are initially empty. All of these should be stored in private attributes. Nothing is returned.

`dealGrid(useFile)` – fills the deck with dominoes from a file (if `useFile` is True) or from a standard "double-six" set (if `useFile` is False). Any exceptions raised while filling the deck should be propagated and the grid should not be populated. If the deck was filled successfully, dominoes are dealt from the deck and added face down to the grid, starting in the upper left corner and moving right until the first row is complete, then moving onto the second row (left to right), third row (left to right), and fourth row (left to right). Nothing is returned.

`select(row, col)` – after asserting that `row` and `col` are valid, removes the domino at the specified row, column position from the grid and replaces it with the string `'***'` (3 asterisks). If there is no domino at the specified location (e.g. if it is already the string `'***'`), raise an Exception with the argument "There is no domino at row xx, column yy", where xx is the specified row and yy is the specified column. Returns the removed domino face up.

`playDomino(domino, stackNum)` – after asserting that `domino` is a Domino and `stackNum` is a valid integer, tries to add the provided `domino` to the top of the stack indicated by `stackNum`, and displays a message describing this action (see sample output). If successfully added, displays "Success!" on the screen and returns True; otherwise, displays the resulting error message and returns False.

`isWinner()` – returns True if one of the stacks contains at least 6 dominoes, False otherwise.

`getNumStacks()` – returns the integer number of stacks in the playing area.

`getNumRows()` – returns the integer number of rows in the grid of dominoes.

`getNumColumns()` – returns the integer number of columns in the grid of dominoes.

`revealGrid()` – displays the face up version of all dominoes that are currently in the grid, for the player to see. Be sure that all dominoes in the grid are face down again before leaving this method. Nothing is returned.

`__str__()` – returns the string representation of the grid and playing stacks on the table. See sample output for formatting.

Be sure to test this Table class thoroughly before moving on. However, you do not need to include these tests for the marker to see.

Task 5: main program

In assignment2.py, prompt the user to select whether s/he wishes to play in Test Mode or Game Mode. Continue to re-prompt until the user enters a valid choice. Create an instance of your Table class. Initialize your table so that its deck is populated from a file if in test mode, or from a shuffled standard "double-six" set if in game mode. If the deck is invalid, the game should end with an explanatory message (see sample **output_invalidDeck1.txt**).

When in test mode:
Before play begins, display all of the dominoes (face up) in your grid. This will make it easy for you to check that you are selecting the correct dominoes as you continue your tests. For the rest of the time in this mode, be sure that you display your grid with the dominoes face down.

Select the domino in the upper left corner of the grid. Attempt to play that domino on a stack, starting with stack 0. If it cannot be played on stack 0, try to play it on stack 1. If it cannot be played on stack 1, try to play it on stack 2. If it cannot be played on stack 2, the player loses. Continue selecting dominoes from the grid and trying to play them on the stacks (as described above). When selecting from the grid, move left to right, then down to the next row, left to right, and so on until the game is over or there are no more dominoes left to select from the grid.

At the end display whether the game was won (if one of the stacks has 6 dominoes in it) or lost (could not play a domino).

When in game mode:
Display "Under Construction" on the screen and end the game.

Task 6: main program (**OPTIONAL** – not for marks, but gives you a more interesting game to play):

When in game mode:
Prompt the user to enter the row and column of the domino s/he would like to select from the grid. If it is a valid location, prompt the user to enter the stack number that s/he would like to try to play the domino on. If it is an empty stack, ask the player which end of the domino should be at the top of the stack, and add the domino according to that specification.

Gracefully handle any errors that result when the player selects an invalid location from the grid, or tries to place a domino on an invalid stack, and continue playing the game until the player wins (has a stack with 6 dominoes on it) or loses (cannot play a domino on any of the stacks).

Sample Output

Refer to the provided sample output files for formatting and exact message displays.

Assessment

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/methods to solve those sub-problems so that each function/method 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 methods and functions; and be sure to acknowledge any collaborators/references in a header comment at the top of your Python files.

Restrictions for this assignment are that you cannot use break/continue, and you cannot import any modules other than the `random` module, the `stack` module (from Lab 5), the `queues` module (from Lab 6), and your `domino175` module. Doing so will result in deductions.

Rubric:

- Code quality and adherence to specifications: 20%
- Domino class and tests: 10%
- DominoDeck class and tests: 15%
- DominoStack class and tests: 15%
- Table class: 25%
- Main automated game: 15%

Submission Instructions

- All of your code should be contained in **TWO** Python files: **domino175.py** and **assignment2.py**.
- Make sure that you include your name (as author) in a header comment at the top of all files, along with an acknowledgement of any collaborators/references.
- Please submit your TWO python files via eClass before the due date/time.
- Do not include any other files in your submission.
- Note that late submissions ***will not be accepted***. You can make as many submissions as you would like before the deadline – only your last submission will be marked. So submit early, and submit often.

REMINDER: Plagiarism will be checked for

Just a reminder that, as with all submitted assessments in this course, we use automated tools to search for plagiarism. In case there is any doubt, you **CANNOT** post this assignment (in whole or in part) on a website like Chegg, Coursehero, StackOverflow or something similar and ask for someone else to solve this problem (in whole or in part) for you. Similarly, you cannot search for and copy answers that you find already posted on the Internet. You cannot copy someone else's solution, regardless of whether you found that solution online, or if it was provided to you by a person you know. **YOU MUST SUBMIT YOUR OWN WORK.**