Team FooBar()

Python-Text-Based-Maze-Game Documentation
Version 2.0

```
def main():
    """
    Function to initialize all necessary variables
    and control the main logic of the application.
    """
```

Game States

```
def show_login_signup_screen():
    """
    Function to display the screen to allow for the user to login and signup.
    """

def show_title_screen():
    """
    Function to display the screen to allow for the user to start
    a new game or watch a replay of an already existing one.
    """

def open_new_game():
    """
    Function to allow for the player to start a new game.
    """
```

Database

```
def password_check(password_input):
    """
    Function to check password for valid input
    :param password_input: input password to check validity of
    :return: help strings for invalid password or valid for good password
    """

def update_top10(new_move_value):
    """
    Function to update and print out the
    top ten moves of the leaderboard.
    :
    """

def save_replays():
    """
    Function to get the 3 most recent replays and save them
    both locally in game files and remotely on Firebase.
    """
```

Text Processing

```
def print_introduction_message():
    """
    Function to print out an introduction message.
    """

def print_replay_message():
    """
    Function to print out a message for the replay system.
    """

def help():
```

```
    """
    Function to print out directions and a list of commands.
    """

def print_input(input):
    """
    Function to get user input from the input text
    :param input: input to print
    """

def print_go_error():
    """
    Function to print error message for invalid go.
    """

def print_input_error():
    """
    Function to print error message for invalid input.
    """

def clear():
    """
    Function to clear user input from the InputText.
    """
```

File Input/Output

```
def manage_log_files():
    """
    Function to manage the number of log files in the current working directory.
    """

def manage_replay_files():
    """
    Function to manage the number of replay files in the directory.
    """

def open_log_file():
    """
    Function to open the log file.
    """

def open_replay_file():
    """
    Function to open the replay file.
    """

def open_chosen_replay_file(number):
    """
    Function to open the replay file chosen by the user.
    :param number: replay number to open
    """

def write_to_log_file(string):
    """
    Function to write to the log file.
    :param string: string to write log file
    """




def write_to_replay_file(string):
    """
    Function to write to the replay file.
    :param string: string to write to replay file
    """
```

```
def close_log_file():
    """
    Function to close the opened log file.
    """

def close_replay_file():
    """
    Function to close the opened replay file.
    """

def close_chosen_replay_file():
    """
    Function to close the chosen replay file.
    """
```

Encryption

```
def database_encrypt(n, plaintext):
    """
    Function to encrypt the database
    :param n:
    :param plaintext: text to encrypt
    :return: encrypted database
    """

def database_decrypt(n, ciphertext):
    """
    Function to decrypt the database
    :param n:
    :param ciphertext: text to decrypt
    :return: decrypted database
    """

def caesar_cipher_encrypt(text, key):
    """
    Function that implements the Caesar Cipher encryption algorithm
    :param text: text to be encrypted
    :param key: encryption key applied to text
    :return: encrypted text
    """

def caesar_cipher_decrypt(text, key):
    """
    Function to decrpyt the Caesar Cipher encrypted text.
    :param text: text to be decrypted
    :param key: decryption key applied to text
    :return: decrypted text
    """
```

class AES(object):

```
    def get_sbox_value(self, num):
    """
    Function to retrieve a given s-box value.
    :param num: index of s-box value
```

```
:return: s-box value
"""
     def get_sbox_invert(self, num):
"""
Function to retrieve a given Inverted S-Box Value.
:param num: index of inverted s-box value
:return: inverted s-box value
"""
     def rotate(self, word):
"""
Function to perform Rijndael's key schedule rotate operation
:param word: word to be rotated
:return: rotated word
"""

     def get_rcon_value(self, num):
"""
Function to retrieve a given Rcon Value
:param num: index of Rcon value
:return: Rcon value
"""

     def core(self, word, iteration):
"""
Function to rotate a word and apply s-box substitution
:param word: word to be modified
:param iteration: loop variable
:return: modified word
"""

     def expand_key(self, key, size, expanded_key_size):

"""
Function to expand the key
:param key: key to be expanded
:param size: expansion size
:param expanded_key_size: final size of the expanded key
:return: expanded key
"""

     def add_round_key(self, state, round_key):
"""
Function to add a round key
:param state:
:param round_key: key added to the state
:return: state
"""

     def create_round_key(self, expanded_key, round_key_pointer):
"""
Function to create a round key from the given expanded key
and the position within the expanded key.
:param expanded_key: given expanded key
:param round_key_pointer:
:return: round key
"""




     def galois_multiplication(self, a, b):
"""
Function to perform Galois multiplication
:param a: 8-bit character
:param b: 8-bit character
:return: product of a x b
"""

     def sub_bytes(self, state, is_inv):
"""
Function to substitute all the values from the state
```

```
with the value in the s-box using the state value
as index for the s-box.
:param state: given state
:param is_inv: bool value to tell if it is inverted or not
:return: state
"""

    def shift_rows(self, state, is_inv):
"""
Function to iterate over the 4 rows and call shift_row() with that row
:param state: state to be shifted
:param is_inv: bool value to tell if it is inverted or not
:return: shifted state
"""

    def shift_row(self, state, state_pointer, nbr, is_inv):
"""
Function that shifts the row to the left by 1 each iteration
:param state: state to be shifted
:param state_pointer:
:param nbr: number for loop variable
:param is_inv: bool value to tell if it is inverted or not
:return: shifted state
"""

    def mix_columns(self, state, is_inv):
"""
Function to perform Galois multiplication of the 4x4 matrix.
:param state: state to mixed (multiplied)
:param is_inv: bool value to tell if it is inverted or not
:return: mixed state
"""


    def mix_column(self, column, is_inv):
"""
Function to perform galois multiplication of 1 column of the 4x4 matrix
:param column: column to mix
:param is_inv: bool value to tell if it is inverted or not
:return: mixed column
"""

    def aes_round(self, state, round_key):
"""
Function that applies the 4 operations of the forward round in sequence.
:param state: state to perform operations on
:param round_key: round key added to the state
:return: state with performed operations applied
"""









    def aes_inv_round(self, state, round_key):
"""
Function that applies the 4 operations of the inverse round in sequence.
:param state: state to perform operations on
:param round_key: round key added to the state
:return: state with performed operations applied.
"""

    def aes_main(self, state, expanded_key, nbr_rounds):
"""
Function to perform the initial operations, the standard round,
and the final operations of the forward aes, creating
a round key for each round.
:param state: state to be modified
:param expanded_key: expanded key to apply to state
```

```
:param nbr_rounds: number of rounds
:return: modified state
"""

    def aes_inv_main(self, state, expanded_key, nbr_rounds):
"""
Function to perform the initial operations, the standard round,
and the final operations of the inverse aes, creating
a round key for each round.
:param state: state to be modified
:param expanded_key: expanded key to apply to state
:param nbr_rounds: number of rounds
:return: modified state
"""

    def encrypt(self, input, key, size):
"""
Function that encrypts a 128 bit input block against the given key of size specified
:param input: input string to encrypt
:param key: encryption key
:param size: size of the key
:return: encrypted string
"""

    def decrypt(self, input, key, size):
"""
Function that decrypts a 128 bit input block against the given key of size specified
:param input: input string to decrypt
:param key: decryption key
:param size: size of the key
:return: decrypted string
"""
```

class AESModeOfOperation(object):

This class handles AES with plaintext consisting of multiple blocks.
Choice of block encoding modes: OFB (Output Feedback),
CFB (Cipher Feedback), CBC (Cipher Block Chaining)

```
    def convert_string(self, string, start, end, mode):
"""
Function to convert a 16 character string into a number array.

"""
```

```
    def encrypt(self, string_in, mode, key, size, IV):
"""
Function to perform Mode of Operation Encryption.
:param string_in: input string
:param mode: mode of type modeOfOperation
:param key: a hex key of the bit length size
:param size: the bit length of the key
:param IV: the 128-bit hex Initialization Vector
:return: Mode of encryption, length of string_in, encrypted string
"""

    def decrypt(self, cipher_in, original_size, mode, key, size, IV):
"""
Function to perform the Mode of Operation Decryption.
:param cipher_in: encrypted string
:param original_size: the unencrypted string length - required for CBC
:param mode: mode of type modeOfOperation
:param key: a number array of the bit length size
```

```
    :param size: the bit length of the key
    :param IV: the 128-bit number array Initialization Vector
    :return: decrypted plain text
    """

        def append_pkcs7_padding(s):
    """
    Function to return s padded to a multiple of 16-bytes by PKCS7 padding.
    :param s: string to apply padding to
    :return: padded string
    """

        def strip_pkcs7_padding(s):
    """
    Function to return s stripped of PKCS7 padding
    :param s: string to strip
    :return: stripped string
    """

        def encrypt_data(key, data, mode=AESModeOfOperation.modeOfOperation["CBC"]):
    """
    Function to encrypt data using the key.
    The key should be a string of bytes.
    :param key: string of bytes
    :param data: data to encrypt
    :param mode: mode of operation of aes encryption
    :return: cipher string prepended with the initialization vector (iv)
    """

        def decrypt_data(key, data, mode=AESModeOfOperation.modeOfOperation["CBC"]):
    """
    Function to decrypt data using key.
    Key should be a string of bytes.
    Data should have the initialization vector (iv) prepended
    as a string of ordinal values.
    :param key: string of bytes
    :param data: data to decrypt
    :param mode: mode of operation of aes decryption
    :return: decrypted data
    """

        def generate_random_key(keysize):
    """
    Function to generate a key from random data of length keysize.
    :param keysize: size of the key
    :return: key as a string of bytes
    """
```

Replay

```
def open_replay(number):
    """
    Function to open and play the replay to the user.
    :param number: replay number
    :return:
    """
```

Rendering

```
def generate_maze_recursive_backtracker():
    """
    Function that generates a random maze
    using the Recursive Backtracker algorithm.
    """
```

```
def generate_maze_binary_tree():
    """
    Function that generates a random maze using the Binary Tree algorithm.
    """

def reset_maze():
    """
    Function to reset the maze back to its original state.
    """

def check_maze_for_validity_player_door():
    """
    Function to test the maze for validity by checking
    the path from the player to the door object.
    :return: 0 if the maze is valid, 1 if the maze is invalid
    """

def check_maze_for_validity_player_key():
    """
    Function to test the maze for validity by checking
    the path from the player to the key object.
    :return: 0 if the maze if valid, 1 if the maze is invalid
    """

def check_maze_for_validity_player_chest():
    """
    Function to test the maze for validity by checking
    the path from the player to the chest object.
    :return: 0 if the maze is valid, 1 if the maze is invalid
    """

def draw_screen(screen):
    """
    Function to draw the screen.
    :param screen: screen to draw
    """

def is_object_visible(object_position_x, object_position_y):
    """
    Function to determine if the given object is within the field of view.
    :param object_position_x: object's x coordinate
    :param object_position_y: object's y coordinate
    :return: bool value
    """



def get_visible_object_list():
    """
    Function to store object coordinates that are within the field of view.
    """

def get_cell_rect(coordinates, screen):
    """
    Function to draw the container of the objects.
    :param coordinates:
    :param screen:
    :return:
    """

def draw_door_object(door_object, screen):
    """
    Function to draw the door object to the console window.
    :param door_object: door object to draw
    :param screen: screen to draw on
    """

def draw_closed_chest_object(chest_object_closed, screen):
    """
    Function to draw the closed chest object to the console window.
    :param chest_object_closed: chest object to draw
```

```
    :param screen: screen to draw on
    """

def draw_opened_chest_object(chest_object_opened, screen):
    """
    Function to draw the opened chest object to the console window.
    :param chest_object_opened: chest object to draw
    :param screen: screen to draw on
    """

def draw_key_object(key_object, screen):
    """
    Function to draw the key object to the console window.
    :param key_object: key object to draw
    :param screen: screen to draw on
    """

def draw_player_object(player_object, screen):
    """
    Function to draw the player character object to the console window.
    :param player_object: player object to draw
    :param screen: screen to draw on
    """

def draw_simple_enemy_object(simple_enemy_object, screen):
    """
    Function to draw the simple enemy object to the console window.
    :param simple_enemy_object: simple enemy object to draw
    :param screen: screen to draw on
    """

def draw_smart_enemy_object(smart_enemy_object, screen):
    """
    Function to draw the smart enemy object to the console window.
    :param smart_enemy_object: smart enemy object to draw
    :param screen: screen to draw on
    """

def draw_chest_combination_1_object(chest_combination_1_object, screen):
    """
    Function to draw the chest combination 1 object to the console window.
    :param chest_combination_1_object: combination to draw
    :param screen: screen to draw on
    """
def draw_chest_combination_2_object(chest_combination_2_object, screen):
    """
    Function to draw the chest combination 2 object to the console window.
    :param chest_combination_2_object: combination to draw
    :param screen: screen to draw on
    """

def draw_chest_combination_3_object(chest_combination_3_object, screen):
    """
    Function to draw the chest combination 3 object to the console window.
    :param chest_combination_3_object: combination to draw
    :param screen: screen to draw on
    """
```

Object Placement

```
def generate_random_object_positions():
    """
    Function to generate random start positions for
    the player, chest, key, door, and enemy objects.
    """

def generate_optimal_object_positions():
    """
    Function to generate random positions for the objects on the optimal path.
```

```
    """

def position_is_object(x, y):
    """
    Function to determine if the coordinate is blocked by an object.
    :param x: object's x coordinate
    :param y: object's y coordinate
    :return: bool value if coordinate is blocked
    """

def position_is_wall(x, y):
    """
    Function to determine if the coordinate is blocked by a wall.
    :param x: wall x coordinate
    :param y: wall y coordinate
    :return: bool value if coordinate is blocked by wall
    """

def reset_object_positions_and_state_conditions():
    """
    Function that resets the position of the player
    and confiscates all gathered items.
    """
```

Player Input

```
def handle_input():
    """
    Function to handle player character movement.
    :return: nothing to return
    """
```

Character Actions

```
def go(dx, dy):
    """
    Function to move the player character object through the maze.
    :param dx: player x coordinate
    :param dy: player y coordinate
    """

def go_length(direction, length):
    """
    Function to move the player character object through
    the maze for the number of times they specify.
    :param direction: direction to move player
    :param length: distance to move player
    """

def use_marker():
    """
    Function to use the marker.
    """

def grab_key():
    """
    Function to grab the key.
    """
```

```
def unlock_chest(user_input_combination):
    """
    Function to unlock the chest.
    :param user_input_combination: combination the user inputs
    """

def open_chest():
    """
    Function to open the chest.
    """

def use_key():
    """
    Function to use the key.
    """

def open_door():
    """
    Function to open the door.
    """

def player_next_to_object(x, y, a, b):
    """
    Function that returns true if the player character
    object is located next to another object.
    :param x: player x coordinate
    :param y: player y coordinate
    :param a: object x coordinate
    :param b: object y coordinate
    :return: bool value if player is next to an object
    """
```

Enemies

```
def move_simple_enemy():
    """
    Function to move the simple enemy object in a random direction.
    """

def move_smart_enemy():
    """
    Function to move the smart enemy in a direction towards the player.
    """

def respawn_smart_enemy():
    """
    Function to reset the smart enemy object position.
    """
```

Pathfinding

class SquareGrid:

Class used to make a graph object for the algorithm.

```
    def __init__(self, width, height):
    """
    Initialize the parameters.
    :param width: width of grid
```

```
:param height: height of grid
"""

    def in_bounds(self, id):
"""
Function to determine if the neighbors are in bounds.
:param id: neighbor id
:return: bool value if neighbor is in bounds
"""

    def passable(self, id):
"""
Function to determine if an element is valid.
:param id: element id
:return: bool value if element is valid
"""

    def neighbors(self, id):
"""
Function to find the neighbors.
:param id: id of item to find neighbors for
:return: resultant list of neighbors
"""
```

class GridWithWeights(SquareGrid, object):

Subclass used to access the cost function.

```
    def __init__(self, width, height):
"""
Function to initialize grid with weights
:param width: width of grid
:param height: height of grid
"""

    def cost(self, from_node, to_node):
"""
Function used to calculate the cost to move from from_node to to_node.
:param from_node: node to move from
:param to_node: node to move to
:return: cost of moving from node to node
"""
```

class PriorityQueue:

Class that associates each item with a priority.

```
    def __init__(self):
"""
Function to initialize.
"""

    def empty(self):
```

```
    """
    Function to empty priority queue
    :return:
    """

        def put(self, item, priority):
    """
    Function to put item in priority queue
    :param item: item to put
    :param priority: priority queue
    """

        def get(self):
    """
    Function to get priority queue
    :return: priority queue elements
    """

def heuristic(a, b):
    """
    Function used to heuristic
    :param a:
    :param b:
    :return:
    """

def a_star_search(graph, start, goal):
    """
    Function that implements the A* algorithm.
    :param graph: the graph we will search
    :param start: the starting location (character location at start)
    :param goal: the ending location (door location at start).
    :return: where we came from and how much it has cost us
    """

def is_valid(point, grid):
    """
    Function to check the validity of a point before it is added as a neighbor.
    :param point: point to check validity of
    :param grid: grid that contains the point
    :return: bool value if point is valid
    """

def add_neighbours(point, neighbours_list, visited_list, grid, dict):
    """
    Function to add all the neighbors of the selected point to the stack (list).
    :param point: point to check neighbors of
    :param neighbours_list: list of neighbors of point
    :param visited_list: list of visited
    :param grid: grid we are using
    :param dict: dictionary of points
    :return: nothing to return
    """

def add_to_dictionary(dictionary, parent, child):
    """
    Dictionary used to print the success path after it has been generated.
    :param dictionary: dictionary we are adding to
    :param parent:
    :param child:
    """

def depth_first_search(start_point, end_point, graph, dict):
    """
    Function that implements the main logic of the DFS algorithm.
    :param start_point: starting point
    :param end_point: ending point
    :param graph: graph we are using
    :param dict: dictionary to add neighbors to
    """
```

```
def breath_first_search(start_point, end_point, graph, dict):
    """
    Function that contains the main logic of the BFS algorithm.
    :param start_point: starting point
    :param end_point: ending point
    :param graph: graph we are using
    :param dict: dictionary to add neighbors to
    """

def draw_hierarchy(dict, point):
    """
    Function to add the locations at the end to the list (stack).
    :param dict: dictionary of points
    :param point: location to add
    :return: list
    """
```