Classes

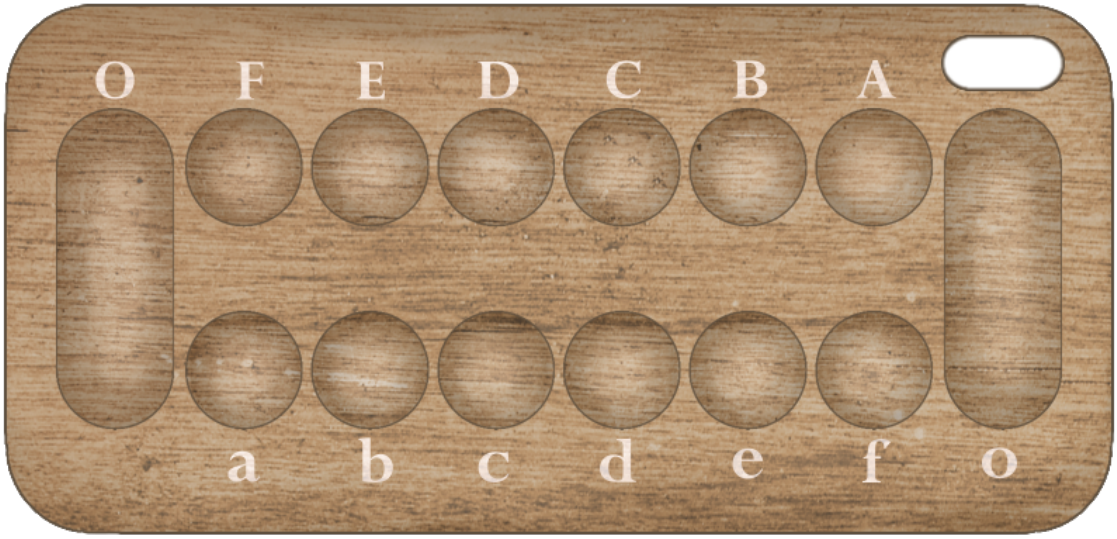
February 17, 2022

```
[ ]: from ipycanvas import MultiCanvas,hold_canvas
from ipywidgets import Image
import math
import random
import IPython.display
import time
import copy
import datetime as dt
from IPython.core.display import HTML
with open('style.css', 'r') as file:
    css = file.read()
HTML(css)
```

1 Kalah Rules

1.0.1 Game structure:

The playing field consists of 6 houses per side and additionally one large house per player at the edge, the so-called storage. At the beginning of the game, six seeds are placed in each house except the storage.



1.0.2 Gameplay:

Alternately, a player selects a house on his side of the board. The player then places the seeds in the following houses in a counter-clockwise direction. If visited, a seed is placed in the own storage, the opponent's storage is left out. The player's turn ends if all seeds are placed.

There are some special rules, that apply if the last seed is placed in the player's storage or an empty player's house: 1. If the last seed is placed in the own storage, it is the player's turn again. 2. If the last seed is placed in an empty house on the player's own side of the board, this seed and all the seeds in the opposite house are placed in the player's storage. In that case, it is the opponent's turn.

1.0.3 End of the game:

The game ends when all the houses of one player have been emptied. The opposing player then places all remaining seed in his storage.

1.0.4 Object of the game:

The winner is the player who has more seeds in his storage at the end of the game.

Source: http://www.kalaha.de/kalaha.htm (9.11.2021)

2 Kalah game definition

2.0.1 Basic definition:

We define the game G as a six-tuple as follows so that a computer is able to play Kalah.

G = &ltStates, s0, Players, nextStates, finished, utility>

The components have the following meanings:

1. **States** is a set that contains all possible states of the game Kalah. A state of the game is represented by a list containing two lists, representing the houses of the two players. The first six values of each list represent the number of seeds in the player's houses represented by the letters **{A, B, C, D, E, F}**. The seventh and last value stands for the number of seeds in the corresponding player's storage. The start state of the game is for example:

[[6,6,6,6,6,6,0], [6,6,6,6,6,6,0]]

2. **s0** **States** is the start state.
3. **Players** is a list that contains the players. Kalah is a game for exactly two players, therefore this game's **Players** list only contains two elements.
4. **nextStates** is a function which calculates a set of states that are reachable by one move from Player p in the state s. To do so, the function receives a state **s** **States** and a player **p** **Players**. The formula is given as follows:

nextStates: States x Players → 2States

5. **finished** is a function that takes a state **s** ε **States** and checks if the game is finished, meaning that one of the players has emptied all their houses . The formula is:
finished: States → B

The function finished is used to compute a set Terminalstates that contains all states from a finished game. This set is defined as follows:
TerminalStates:= {s States | finished(s)}

6. **utility** is a function that calculates the value of the game for a player. Therefore, the function takes a state **s TerminalStates** and a player **p Players**. The value that the function returns is an element from the set **{-1, 0, 1}**. The player p has lost the game when the function returns -1. If the function returns 1, the player has won the game and if the value is 0, the game ends in a draw. The formula for the function is:

utility: TerminalStates x Players → {-1, 0. 1}

source: https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Lecture-Notes/artificial-intelligence.pdf, S.87, Abruf am 06.02.2022

3 Classes

The Kalah game is implemented with the use of several classes, containing base class Kalah_Game, a Board class for displaying the UI and a Player class as well as several classes inheriting from this Player class:

- Human
- Repeated_Player
- Random_AI

The following illustration shows a class diagram of all classes involved:

This image was created with Creatly.

3.1 Global Variables

Additionally to the classes, there is one global variable *gOrder* with the purpose to help converting the state of the game board, which is represented by a nested list, to letters. The letter representation is mainly for the UI, but also distributes to print and logging messages being easier to humanly comprehend.

[ ]:
```
gOrder = [['A','B','C','D','E','F','O'],
          ['a','b','c','d','e','f','o']]
```

3.2 Global Game Functions

- *other_player(player_num)*
- *move(state, player_num, choice)*
- *next_states(state, player_num)*
- *finished(state)*
- *utility(state, player_num)*
- *value(state, player_num)*

TODO

3.2.1 Function: other_player

The function *other_player(player_num)* returns the opponent's player number to a given player number. This means that it returns 1 for the input 0, and 0 for the input 1.

[ ]:
```
def other_player(player_num):
    return (player_num + 1) % 2
```

3.2.2 Function: move

The function *move(choice)* receives a state, the number of a player and the house index they have chosen as a result of the Player method *choose_house*. It calculates the actions of the player's turn and returns the resulting new game state. Additionally, the function is used in *next_states* to calculate all possible following states from one state.

The calculations are implemented based on the following game rules:

1. The current player choses one of his house. The seeds from the chosen house are placed counterclockwise in the house of both players and in the store of the current player. The store of the opponent is left out. Then it is the turn of the opponent.
2. If the last seed is placed in the store of the current player, they get another move.
3. If the last seed is placed in an empty house of the current player the seed from this house and all seeds from the opposite house are place in the store of the current player. Then it's the opponent's turn.

[ ]:
```
def move(state, player_num, choice):
    new_state = copy.deepcopy(state)

    seeds = new_state[player_num][choice]
    new_state[player_num][choice] = 0

    c_house_player_num = player_num
    c_house_num = choice

    # Go through houses counterclockwise
    while (seeds > 0):
        c_house_num += 1

        # Skip opponent's store
        if(c_house_num == 6 and c_house_player_num != player_num):
            continue

        # Switch to houses of the other player
        if(c_house_num > 6):
            c_house_num = 0
            c_house_player_num = other_player(c_house_player_num)

        # Add seed to the currently visited house
```

```
        new_state[c_house_player_num][c_house_num] += 1
        seeds -= 1

    another_turn = False
    # Check for special rules after last seed was placed in own store or own␣
↪empty house:
    if(c_house_player_num == player_num):

        # Rule: Another turn if last seed is placed in own store
        if(c_house_num == 6):
            # Give current player another turn
            another_turn = True

        # Rule: Last seed is placed in empty house of current player
        elif(new_state[player_num][c_house_num] == 1):
            # Collect all seeds to be rewarded and empty both houses
            receivedSeeds = new_state[player_num][c_house_num]
            receivedSeeds += new_state[other_player(player_num)][5 -␣
↪c_house_num]
            new_state[other_player(player_num)][5 - c_house_num] = 0
            new_state[player_num][c_house_num] = 0
            # Award all the seeds to the current player's kalah
            new_state[player_num][6] += receivedSeeds
            # if self.display_mode != 0:
                # print(f'Player {self.players[player_num].name} gets a steal␣
↪for {receivedSeeds} Seeds!')

    move_log = tuple([player_num, choice, copy.deepcopy(new_state)])

    return new_state, another_turn, move_log
```

3.2.3 Function: next_states

The function *nextStates()* is a function which calculates a set of states that are reachable by one move from Player p in the state s. To do so, the function receives a state **s** **States** and a player **p** **Players**. The formula is given as follows:

nextStates: States x Players → 2States

[ ]:
```
def next_states(state, player_num):
    states = []
    for choice in range(6):
        # Check if choice is valid (at least one seed in house)
        if state[player_num][choice] == 0:
            continue
        next_state, another_turn, _ = move(state, player_num, choice)
```

```
        # Check if player has another turn
        # If so, next_states is called recursively until the player has no␣
↪other turn
        if another_turn:
            for s,choices in next_states(next_state, player_num):
                states.append((s,[choice] + choices))
        else:
            states.append((next_state,[choice]))

    return states
```

[ ]:
```
next_states([[3, 4, 1, 2, 0, 7, 19], [5, 2, 2, 1, 2, 1, 23]],0)
```

3.2.4 Method: finished

The private method *finished()* checks if one of the players has no seeds in their house left and is therefore unable to take another turn. If this is the case, the method returns True, otherwise it returns False.

[ ]:
```
def finished(state):
    sum0 = sum(state[0][:-1])
    sum1 = sum(state[1][:-1])

    if(not (sum0 == 0 or sum1 == 0)):
        return False
    return True
```

3.2.5 Method: utility

The method *utility(player_num)* receives the number of the current player and uses the current state to calculate the utility of the state for the player. The method compares the seeds from both stores and returns an Integer dependent on the result. If the current player has more seeds than the opponent the method returns 1. If they own less seeds than the opponent the method returns -1 and if it is a draw the number 0 is returned.

[ ]:
```
def utility(state, player_num):
    playerStore = state[player_num][6]
    opponentStore = state[other_player(player_num)][6]
    if(playerStore > opponentStore):
        return 1
    elif(playerStore == opponentStore):
        return 0
    else:
        return -1
```

3.2.6 Method: value

TODO!

[ ]:
```
def to_tuple(state_list):
    return tuple(tuple(s) for s in state_list)
```

[ ]:
```
to_tuple([[1, 8, 1, 0, 1, 2, 16], [1, 4, 0, 4, 15, 14, 5]])
```

[ ]:
```
def to_list(state_tuple):
    return list(list(s) for s in state_tuple)
```

[ ]:
```
to_list([[1, 8, 1, 0, 1, 2, 16], [1, 4, 0, 4, 15, 14, 5]])
```

[ ]:
```
Cache = {}

def memoize(f):
    global Cache

    def f_memoized(*args):
        args = (to_tuple(args[0]),args[1],args[2])
        if args in Cache:
            return Cache[args]

        result = f(to_list(args[0]),args[1],args[2])
        Cache[args] = result
        return result

    return f_memoized
```

[ ]:
```
@memoize
def value(state, player_num, limit):
    if finished(state) or next_states(state, player_num) == [] or limit==0:
        return utility(state, player_num)
    other = other_player(player_num)
    limit -= 1
    return max([-value(ns, other,limit) for ns,_ in next_states(state,␣
 ↪player_num)])
```

[ ]:
```
#print(value([[1, 8, 1, 0, 1, 2, 16], [1, 4, 0, 4, 15, 14, 5]],0,4))
```

4 Player Class

Attributes:

- *player_type*
- *number*
- *name*

Methods:

- *___init___(number, name)*
- *___str___()*
- *_available_house(own_house)*
- *choose_house(current_state)*

The class Player is the superclass which represents a Kalah game player. In Kalah there are exactly two players which compete against each other.

The attribute *player_type* is only for subclasses of the Player class. It contains a string representation of the name of the Player subclass. This makes it possible to print the specific type of a Player object. This is for example used for creating log files.

The attribute *number* of a player has either the value 0 or 1 depending on the order in which the players take their turns. If the number of the player is 0, they start the game by having the first turn. Additionally, *number* represents the index of the state list which contains the player's list of house values.

The attribute *name* is a string which represents the player's name in the UI or print and logging messages.

4.0.1 Method: init

The method *___init___(number, name)* initializes the Player object and checks if the given *number* is either 0 or 1 and if the given *name* is a string. If one of these criteria is not met, an error message is raised.

4.0.2 Method: str

The method *___str___()* defines the *name* of the Player object as their string representative for print messages.

[ ]:
```
class Player:

    def __init__(self, number, name):

        if number in [0,1]:
            self.number = number
        else:
            raise ValueError("Number of player must be 0 or 1!")
        if isinstance(name, str):
            self.name = name
        else:
            raise ValueError("Name must be a string!")

    def __str__(self):
        return self.name
```

4.0.3 Method: _available_house

The private method *_available_house(own_house)* receives a list which represents the player's house and their store. The method returns a list with the indizes of all house which contain at

least one seed (list indizes with a value higher than null).

[ ]:
```
def _available_house(self, own_house):
    available = []
    for i in range(len(own_house)):
        if own_house[i] != 0:
            available.append(i)
    return available

Player._available_house = _available_house
```

4.0.4 Method: choose_house

The method *choose_house(current_state)* receives the current state of the Kalah board and returns the index of the chosen house from the player's house list. This method differentiates the Player subclasses and is therefore differently implemented in each of them. There is no basic implementation of this method in the Player class. This is why in the class Kalah_Game only subclasses of Player are accepted as players. Therefore, this class is intended as an abstract class.

[ ]:
```
def choose_house(self, current_state):
    pass

Player.choose_house = choose_house
```

4.1 Tests of Player Class

4.1.1 1. Player number must be 0 or 1

[ ]:
```
try:
    Player(2, "Player1")
except ValueError as e:
    print(e)
```

4.1.2 2. Player name must be a string

[ ]:
```
try:
    Player(0, 3)
except ValueError as e:
    print(e)
```

4.1.3 3. Successful Player creation

[ ]:
```
Player(0, "Player1")
```

4.2 Human Player Class

The class Human inherits from Player. It is used for humans playing against each other or against one of the AIs.

[ ]:
```
class Human(Player):

    def __init__(self, number, name):
        self.player_type = "Human"
        super().__init__(number, name)
```

4.2.1 Method: choose_house

The method *choose_house(current_state)* for the Human class is implemented as follows:

At first the own house are extracted from the game state and the available house indizes are calculated using the method *_available_house*. Afterwards the human player is asked to choose one of the available house via a input field. With limiting the player to the available house, it is easier to detect a correct input. If the player enters an unvalid value, they are asked to input a value again until a valid value is given. The index of the corresponding house is returned.

[ ]:
```
def choose_house(self, current_state):
    own_t,store = current_state[self.number][:6],current_state[self.number][6]
    available = self._available_house(own_t)

    i_string = "Choose one of the available houses:\n"
    for i in available:
        i_string += f"{gOrder[self.number][i]}, "

    choice_str = ""
    letter_numbers = {gOrder[1][i]:i for i in range(6)}

    while choice_str not in [k for k in letter_numbers if letter_numbers[k] in␣
 ↪available]:
        choice_str = input(i_string[:-2]+"\n").lower()

    choice = letter_numbers[choice_str]
    IPython.display.clear_output()
    return choice

Human.choose_house = choose_house
```

4.2.2 Test of Human Player Class

[ ]:
```
human = Human(1,"Hans")
```

[ ]:
```
#human.choose_house([[6,6,6,6,6,6,0], [6,0,6,6,4,6,0]])
```

4.3 Repeated Player Class

The class Repeated_Player inherits from Player. It is used for repeating the behavior of a player from a previous game on basis of a given log file.

For this reason it takes the **logged_moves** information from the logged game and saves it as *moves_to_repeat*, so that these moves can be replicated in the *choose_house* method.

[ ]:
```
class Repeated_Player(Player):

    def __init__(self, number, name, logged_moves):
        self.player_type = "Repeated_Player"
        self.moves_to_repeat = logged_moves
        super().__init__(number, name)
```

4.3.1 Method: choose_house

The method *choose_house(current_state)* for the Repeated_Player class is implemented as follows:

The list *moves_to_repeat* is handled as a stack in this method. This means that one after one, the moves which stand at the beginning of the list are extracted and removed from the list. In the beginning of the method this process is repeated until the current state of the game matches the state of the last removed move from the *moves_to_repeat* list. This means, that the move which the player will do next is should now be at the beginning of the list.

In the next step, this move is also extracted from the list, but not removed as it could contain the current state for the next move of the player if they gain another turn. To make sure, that this move was actually done by this player in the logged game, the logged move player number is compared to the actual player number. If the numbers match, the choice from that move is returned.

[ ]:
```
def choose_house(self, current_state):

    state = self.moves_to_repeat.pop(0)[2]
    while state != current_state:
        if len(self.moves_to_repeat) == 0:
            raise ValueError("There is no move left to repeat!")
        state = self.moves_to_repeat.pop(0)[2]

    this_move = self.moves_to_repeat[0]
    if this_move[0] not in (-1,self.number):
        raise ValueError("This move was originally not played by this player!")

    choice = this_move[1]

    return choice

Repeated_Player.choose_house = choose_house
```

4.4 Random_AI Player Class

The class Random_AI inherits from Player. It is a simple AI player implementation which chooses house at random.

The class Random_AI has a modified *___init___* function which has an additional argument *seed* for setting a seed for the **random** library functions. Whenever the same seed is set, the random function creates the same random numbers or chooses the same random objects from a list. Adding this argument eases the logging and debbuging of Random_AI players.

[ ]:
```
import random as rn

class Random_AI(Player):

    def __init__(self, number, name, seed):
        self.player_type = "Random_AI"
        self.seed = seed
        rn.seed(seed)
        super().__init__(number, name)
```

4.4.1 Method: choose_house

The method *choose_house(current_state)* for the Random_AI class is implemented as follows:

At first the own house are extracted from the game state and the available house indizes are calculated using the method *_available_house*. From the list of available house indizes, one is chosen at random using the function **choice** from the library **random**. Afterwards, this value is returned.

[ ]:
```
def choose_house(self, current_state):
    own_t,safe = current_state[self.number][:6],current_state[self.number][6]
    available = self._available_house(own_t)

    choice = rn.choice(available)

    return choice

Random_AI.choose_house = choose_house
```

4.4.2 Test of Random_AI Player Class

[ ]:
```
ai = Random_AI(0,"Rando", 1)
```

[ ]:
```
ai.choose_house([[6,6,6,6,6,6,0], [6,6,6,6,6,6,0]])
```

5 Minmax Player Class

TODO: Put in Player Class and rework for that matter

[ ]:
```
class Minmax(Player):

    def __init__(self, number, name, limit, seed):
        self.player_type = "Minmax"
        self.limit = limit
        self.seed = seed
        rn.seed(seed)
        super().__init__(number, name)
```

5.0.1 Method: choose_house

TODO

[ ]:
```
def choose_house(self, current_state):

    NS = next_states(current_state, self.number)
    bestVal = value(current_state, self.number,self.limit)

    BestChoices = [choices[0] for state,choices in NS if -value(state,␣
↪other_player(self.number),self.limit-1) == bestVal]
    choice = random.choice(BestChoices)

    if choice == None:
        raise ValueError(f'No choice for Minmax found.\nbestVal:␣
↪{bestVal}\nBestChoices: {BestChoices}\nnext_states: {NS}')

    return choice

Minmax.choose_house = choose_house
```

[ ]:
```
Minmax(1,"Mini",3,2).choose_house([[6,6,6,6,6,6,0], [6,6,6,6,6,6,0]])
```

6 Kalah_Game Class

Attributes:

- *state*
- *players*
- *current_player*
- *display_mode*
- *logged_moves*

Methods:

- *___init___(players)*
- *_show_state(state)*
- *show_state()*
- *draw_board()*

- *_draw_seeds()*
- *_draw_numbers()*
- *start()*
- *log_to_file(file_id)*

The class Kalah_Game is the core of the Kalah game implementation. It contains all information on the game, including the current game state.

The attribute *state* represents the state of the Kalah board which is defined by the number of seeds laying in each of the player's house and their stores. It is implemented by a nested list which contains a list for the house on each of the two players' sides. The last index of each of the lists is the store of that player. At the start of the game, where the stores are empty and there are six seeds in every house, the implemented representation of the state for example is: [[6,6,6,6,6,6,0], [6,6,6,6,6,6,0]].

The attribute *players* is a list of the two players that play the game. They must be instances of a subclass of the Player class. The order in which they take turns is determined by their Player *number*: The player with the number 0 goes first.

The attribute *current_player* has either the value 0 or 1. It takes track of which player's turn it currently is. If *current_player* has the value 0 for example, it is the turn of the player which stands at index 0 of the *players* list and therefore also has the Player's class attribute *number* of value 0.

The attribute *display_mode* defines which way the game is displayed in the output console. The possible values are 0, 1 and 2 and are defined as follows: - 0: No output is displayed. The game is only logged to the log file - 1: The board is only displayed using the print method *_show_state - *2: The board is displayed with the function* draw_board* using **ipycanvas** for rendering

The attribute *logged_moves* is a list containing a tuple for every move that is made in the game. This list is necessary for the method *log_to_file*. Each tuple has the player number of the player that have made the move at the first index, the index of the chosen house at the second index and the resulting game state as the third index. The start state is saved as the first move in the list, having -1 as the player number and -1 as the chosen house number. As Python uses references, the states are saved in this list by creating deep copies with the library **copy**. At the beginning of the game *logged_moves* looks like this:

[(-1, -1, [[6,6,6,6,6,6,0], [6,6,6,6,6,6,0]])]

6.0.1 Method: init

The method *___init___(players)* initializes the game by setting the *state* to the initial state (seen above), initializing a board object and setting the *players* list using the received "players" argument. Before setting the *players* list, the received list is checked for the number of items it contains (must be exactly 2) and if the items in the list are both instances of a Player subclass (but not of the class Player itself). In error case, matching error messages are raised.

[ ]:
```
class Kalah_Game():

    def __init__(self, players, display_mode):
        self.state = [[6,6,6,6,6,6,0], [6,6,6,6,6,6,0]]
        self.turn = 0
```

```
        if len(players) != 2:
            raise ValueError("There must be exactly two players!")
        if not ((isinstance(players[0], Player) and type(players[0]) != Player)
            and (isinstance(players[1], Player) and type(players[1]) !=␣
↪Player)):
            raise ValueError("Both players must be of instances of a subclass␣
↪of the class Player!")
        if {players[0].number, players[1].number} != {0,1}:
            raise ValueError("One of the players must be self.number 0 and the␣
↪other one self.number 1!")

        self.players = players
        self.current_player = 0

        if display_mode not in range(3):
            raise ValueError("The display mode must be 0, 1 or 2!")
        self.display_mode = display_mode

        self.logged_moves = [(-1,-1,copy.deepcopy(self.state))]
```

6.0.2 Methods: _show_state and show_state

The private method *_show_state(state)* creates a formatted string which represents the received state and prints it to the console. It can be used as an alternative to the **ipycanvas** game UI.

The method *show_state()* calls the private method *_show_state(state)* with the current game state (attribute *state*).

[ ]:
```
def _show_state(self, state):
    s = f''

    s += f'{self.players[0].name}:\t\t\t'
    for j in range(6,-1,-1):
        s += f'{gOrder[0][j]}: {state[0][j]}  '
    s += f'\n'

    s += f'{self.players[1].name}:\t\t\t'
    for j in range(7):
        s += f'{gOrder[1][j]}: {state[1][j]}  '
    s += f'\n'

    print(s)

Kalah_Game._show_state = _show_state
```

[ ]:
```
def show_state(self):
    self._show_state(self.state)

Kalah_Game.show_state = show_state
```

6.0.3 Method: draw_board

The method *draw_board()* sets the required positional values for the methods *_draw_seeds()* and *_draw_numbers()*.

It also translates the current state of the game to a dictionary so that the positional values can be related to the corresponding seed amount in the respective houses.

Additionally this function creates a MultiCanvas for the display of the board. The first layer is the background and the second layer is the foreground.

Afterwards this function calls the functions *_draw_seeds()* and *_draw_numbers()*.

Finally the finished canvas gets displayed.

[ ]:
```
def draw_board(self):
    TOP_ROW_POSITION = (175,130)
    HOUSE_OFFSETS = (90,145)
    HOUSE_MAP={
        'F':TOP_ROW_POSITION,
        'E':(TOP_ROW_POSITION[0]+HOUSE_OFFSETS[0],TOP_ROW_POSITION[1]),
        'D':(TOP_ROW_POSITION[0]+HOUSE_OFFSETS[0]*2,TOP_ROW_POSITION[1]),
        'C':(TOP_ROW_POSITION[0]+HOUSE_OFFSETS[0]*3,TOP_ROW_POSITION[1]),
        'B':(TOP_ROW_POSITION[0]+HOUSE_OFFSETS[0]*4,TOP_ROW_POSITION[1]),
        'A':(TOP_ROW_POSITION[0]+HOUSE_OFFSETS[0]*5,TOP_ROW_POSITION[1]),
        'a':(TOP_ROW_POSITION[0],TOP_ROW_POSITION[1]+HOUSE_OFFSETS[1]),
        'b':
↪(TOP_ROW_POSITION[0]+HOUSE_OFFSETS[0],TOP_ROW_POSITION[1]+HOUSE_OFFSETS[1]),
        'c':
↪(TOP_ROW_POSITION[0]+HOUSE_OFFSETS[0]*2,TOP_ROW_POSITION[1]+HOUSE_OFFSETS[1]),
        'd':
↪(TOP_ROW_POSITION[0]+HOUSE_OFFSETS[0]*3,TOP_ROW_POSITION[1]+HOUSE_OFFSETS[1]),
        'e':
↪(TOP_ROW_POSITION[0]+HOUSE_OFFSETS[0]*4,TOP_ROW_POSITION[1]+HOUSE_OFFSETS[1]),
        'f':
↪(TOP_ROW_POSITION[0]+HOUSE_OFFSETS[0]*5,TOP_ROW_POSITION[1]+HOUSE_OFFSETS[1]),
    }

    canvas = MultiCanvas(2,width=800, height=400)
    canvas[0].draw_image(Image.from_file('images/Board.png'),0,0)

    with hold_canvas(canvas):
        self._draw_seeds(canvas[1],HOUSE_MAP)
        self._draw_numbers(canvas[1],HOUSE_MAP)
```

```
    display(canvas)

Kalah_Game.draw_board = draw_board
```

6.0.4 Method: __draw__seeds

The method *__draw__seeds()* reads the game of the state and accordingly draws seeds on the respective houses. Each seed gets drawn in a random position inside the house with an additional random rotation.

To place the seeds accurately we must consider the width and height of the image of one seed. To do so we subtract the width and height from the final position chosen.

[ ]:
```
def _draw_seeds(self,canvas,housemap):
    HOUSE_RAD = 23
    HOUSE_OFFSETS = (90,125)
    seedsprites = [Image.from_file(f'images/Samen{i}.png') for i in range(1,6)]
    seed_width = 15
    seed_height = 15

    gamestate = self._state_to_dict()
    for house in gamestate:
        for seed in range(gamestate[house]):
            seed_sprite = seedsprites[random.randrange(5)]

            if house == 'O':
                centerX, centerY = (housemap['F'][0]-HOUSE_OFFSETS[0],random.
↪randrange(housemap['F'][1],housemap['F'][1]+HOUSE_OFFSETS[1]))
            elif house == 'o':
                centerX, centerY = (housemap['F'][0]+HOUSE_OFFSETS[0]*6,random.
↪randrange(housemap['F'][1],housemap['F'][1]+HOUSE_OFFSETS[1]))
            else:
                centerX, centerY = housemap[house]

            r = HOUSE_RAD * math.sqrt(random.random())
            theta = random.random() * 2 * math.pi
            x = centerX + r * math.cos(theta)
            y = centerY + r * math.sin(theta)
            canvas.save()

            #Rotate by random amount for variation
            canvas.translate(x, y)
            canvas.rotate(random.uniform(0.,math.pi))
            canvas.translate(-x, -y)
            canvas.draw_image(seed_sprite, x-seed_width, y-seed_height)
            canvas.restore()
```

```
Kalah_Game._draw_seeds = _draw_seeds
```

6.0.5 Method: __draw__numbers

The method *_draw_numbers()* reads the game of the state and accordingly draws numbers that represent the amount of seeds on the respective houses.

[ ]:
```
def _draw_numbers(self,canvas,housemap):
    NUMBER_OFFSET_Y_BOT = 50
    NUMBER_OFFSET_Y_TOP = 60
    HOUSE_OFFSETS = (90,145)
    housenumbermap = {
        'F': (housemap['F'][0],housemap['F'][1]+NUMBER_OFFSET_Y_TOP),
        'E': (housemap['E'][0],housemap['E'][1]+NUMBER_OFFSET_Y_TOP),
        'D': (housemap['D'][0],housemap['D'][1]+NUMBER_OFFSET_Y_TOP),
        'C': (housemap['C'][0],housemap['C'][1]+NUMBER_OFFSET_Y_TOP),
        'B': (housemap['B'][0],housemap['B'][1]+NUMBER_OFFSET_Y_TOP),
        'A': (housemap['A'][0],housemap['A'][1]+NUMBER_OFFSET_Y_TOP),
        'a': (housemap['a'][0],housemap['a'][1]-NUMBER_OFFSET_Y_BOT),
        'b': (housemap['b'][0],housemap['b'][1]-NUMBER_OFFSET_Y_BOT),
        'c': (housemap['c'][0],housemap['c'][1]-NUMBER_OFFSET_Y_BOT),
        'd': (housemap['d'][0],housemap['d'][1]-NUMBER_OFFSET_Y_BOT),
        'e': (housemap['e'][0],housemap['e'][1]-NUMBER_OFFSET_Y_BOT),
        'f': (housemap['f'][0],housemap['f'][1]-NUMBER_OFFSET_Y_BOT),
        'O':␣
↪(housemap['F'][0]-HOUSE_OFFSETS[0],housemap['F'][1]+HOUSE_OFFSETS[1]+HOUSE_OFFSETS[1]/
↪5),
        'o':␣
↪(housemap['F'][0]+HOUSE_OFFSETS[0]*6,housemap['F'][1]+HOUSE_OFFSETS[1]+HOUSE_OFFSETS[1]/
↪5),

    }
    TURN_POS = (housemap['A'][0]+HOUSE_OFFSETS[0],62)

    gamestate = self._state_to_dict()
    canvas.font = '26px serif'
    canvas.text_align    = 'center'

    for key in housenumbermap:
        canvas.
↪fill_text(gamestate[key],housenumbermap[key][0],housenumbermap[key][1])

    canvas.font = '22px serif'
    canvas.fill_text('Turn '+str(self.turn),TURN_POS[0],TURN_POS[1])


Kalah_Game._draw_numbers = _draw_numbers
```

6.0.6 Method: __state__to__dict

The private method *_state_to_dict()* transforms a gamestate into a dictionary.

[ ]:
```
def _state_to_dict(self):
        d = {}
        for i in [0,1]:
            for j in range(7):
                d[gOrder[i][j]] = self.state[i][j]
        return d
Kalah_Game._state_to_dict = _state_to_dict
```

6.0.7 Method: start

The method *start()* starts the Kalah game. Until the game is finished (the method *_finished()* returns True), both players take turns, starting with the player with the number 0. At the start of each turn, the current game state is shown. Next, the current player chooses one of the house with the Player method *choose_house(current_state)*. Afterwards, this choice is handed to the private method *_move(player_num, choice)* which calculates the new game state. The attribute *state* is updated with this new game state. Then it is the turn of the other player. If the game is finished, the method *utility()* calculates which of the players wins the game and the result is printed to the console.

[ ]:
```
def start(self):
    while(not finished(self.state)):
        self.turn += 1
        if self.display_mode == 1:
            print("\nCurrent state:")
            self.show_state()
            print(f"Next is {self.players[self.current_player].name}'s turn.")
        elif self.display_mode == 2:
            print("Current state:")
            self.draw_board()
            print(f"Next is {self.players[self.current_player].name}'s turn.")
            time.sleep(0.05)

        choice = self.players[self.current_player].choose_house(self.state)

        if self.display_mode != 0:
            print(f"{self.players[self.current_player].name} chose {gOrder[self.
↪current_player][choice]}")

        self.state, another_turn, move_log = move(self.state, self.
↪current_player, choice)
        self.logged_moves.append(move_log)

        if not another_turn:
            self.current_player = other_player(self.current_player)
```

```
        elif another_turn and self.display_mode != 0:
            print(f"{self.players[self.current_player].name} gets another turn!
↪")

    won0 = utility(self.state,0)

    if self.display_mode != 0:
        if self.display_mode == 1:
            print("")
            self.show_state()
        elif self.display_mode == 2:
            self.draw_board()

        print("Finished Game!")

        if(won0 == 1):
            print(f"{self.players[0]} wins!")
        elif(won0 == -1):
            print(f"{self.players[1]} wins!")
        else:
            print(f"Draw!")

Kalah_Game.start = start
```

6.0.8 Method: log_to_file

The method *log_to_file* creates a json file from all important game information, including the player subclass types and names as well as the different moves from the game and the winner(s). With the use of this method, played games can be saved and analyzed. The created log file is saved to the folder "logs" with the given file_id in the file name.

[ ]:
```
import json

def log_to_file(self, file_id):
    json_dict = {
        "players":{
            0:{},
            1:{}
        },
        "moves":self.logged_moves,
        "winner":[i for i in range(2) if utility(self.state,i) != -1]
    }

    for i in range(2):
        json_dict["players"][i]["type"] = self.players[i].player_type
        json_dict["players"][i]["name"] = self.players[i].name
        if isinstance(self.players[i], Random_AI):
```

```
            json_dict["players"][i]["seed"] = self.players[i].seed
        if isinstance(self.players[i], Minmax):
            json_dict["players"][i]["seed"] = self.players[i].seed
            json_dict["players"][i]["limit"] = self.players[i].limit

    with open(f"logs/log_{file_id}.json", "w") as f:
        f.write(json.dumps(json_dict, indent=4))
        f.close()

Kalah_Game.log_to_file = log_to_file
```

6.1 Tests of Kalah_Game Class

6.1.1 1. There must be exactly two players

[ ]:
```
try:
    game = Kalah_Game([Player(0,"Test")],0)
except ValueError as e:
    print(e)
```

6.1.2 2. Players must be instances of a Player sublass

[ ]:
```
try:
    game = Kalah_Game([Player(0,"Test"), Player(1,"Test")],0)
except ValueError as e:
    print(e)
```

6.1.3 3. Player numbers must be 0 and 1

[ ]:
```
try:
    game = Kalah_Game([Human(1,"Human"), Random_AI(1,"AI",2)],0)
except ValueError as e:
    print(e)
```

6.1.4 4. Display mode must be 0, 1 or 2

[ ]:
```
try:
    game = Kalah_Game([Human(1,"Human"), Random_AI(0,"AI",2)],3)
except ValueError as e:
    print(e)
```

6.1.5 5. Successful Game creation with two AIs

[ ]:
```
game = Kalah_Game([Random_AI(0,"Rando",1), Minmax(1,"Mini",3,2)],1)
```

[ ]:
```
game.show_state()
```

```
[ ]: %%time
     game.start()
```

```
[ ]: timestamp = str(dt.datetime.now()).replace(':','.')
     game.log_to_file(timestamp)
```

6.2 Repeat Game from Log File

The function *repeat_game* takes the filename of a log file and the display mode with which the game should be repeated as the input values. It also receives the information, if included AIs should calculate their choices new or if they should just repeat the moves that are presented in the log file. The decicions of human players are always repeated based on the log file.

The function detects the player types and names from the file and creates players based on this information. The player types are only relevant for AIs if **repeat_AIs** is set to False. In all other cases, the players are created from the Repeated_Player class.

In the end, the game is initialized with the created players and the given display mode and then started.

```
[ ]: def repeat_game(filename, repeat_AIs, display_mode):
         json_dict = {}

         with open("logs/"+filename) as f:
             json_dict = json.load(f)

         players = []
         for i in range(2):
             i = str(i)
             p_type = json_dict["players"][i]["type"]

             if repeat_AIs:
                 players.
     ↪append(Repeated_Player(int(i),json_dict["players"][i]["name"],json_dict["moves"]))
             else:
                 if p_type == "Human":
                     players.
     ↪append(Repeated_Player(int(i),json_dict["players"][i]["name"],json_dict["moves"]))
                 elif p_type == "Random_AI":
                     players.
     ↪append(Random_AI(int(i),json_dict["players"][i]["name"],json_dict["players"][i]["seed"]))
                 elif p_type == "Minmax":
                     players.
     ↪append(Minmax(int(i),json_dict["players"][i]["name"],json_dict["players"][i]["limit"],json_
                 else:
                     raise ValueError("The given player type is unknown!")

         game = Kalah_Game([players[0],players[1]],display_mode)
```

```
    game.start()
```

6.2.1 Tests of Repeat Game

Repeating the game of two AIs, where the decisions are repeated based on the log file (Player types: Repeated_Player)

[ ]:
```
%%time
repeat_game(f"log_{timestamp}.json",True,1)
```

Repeating the game of two AIs, where both of them calculate their decisions new (Player types: Random_AI)

[ ]:
```
%%time
repeat_game(f"log_{timestamp}.json",False,1)
```

6.3 Play against Random AI

[ ]:
```
gameHuman = Kalah_Game([Random_AI(0,"AI2",1), Human(1,"Karl")],2)
```

[ ]:
```
# gameHuman.start()
```

Created in Deepnote