SWE-Skat

version

Pascal Stehling

Februar 10, 2020

Contents

Welcome to SWE-Skat's documentation!

This is my little Petproject in which I programmed the card game Skat. The program itself is usable, but does not contain all game elements. So it is for example not yet possible to make a zero game. The goal of this project was not to program a perfect Skat, but to try out many different tools and paradigms around programming.

The game is written in pure Python and does not need any other libraries to run it. It can be started inside the root folder with the command python main.py.

In the next chapters you will find all points concerning the individual paradigms/tools.

Build Management

The main build management tool used was Bazel. With this tool it is possible to build a Skat.exe and execute the tests. In order to use Bazel, a BUILD.bazel file must be written, which contains all the necessary information about the individual files and the relationships between them. This file can be found in the root folder of this project. To build the .exe the command *bazel build Skat* can be executed in the root folder. This will create a Skat.exe in the bazel-bin folder. To run the tests, the command *bazel run test* can be used. For more information on installing Bazel, please visit their website.

For further automation the tool doit can also be used. With this tool all unit tests can be executed with the command *doit run_test*. In addition, a coverage.xml is created which contains all information about the coverage of the code. Furthermore, with the command *doit lint_modules* all files in the folder modules can be viewed with pylint. With the command *doit bazel_build* the Skat.exe can be built with bazel. To use doit you just have to install all the required libraries with *pip install -r requirements.txt*.

Clean Code

In this project I tried to keep the code as clean as possible and as easy to understand as possible. Therefore the code was refactored and rewritten very often which can be seen in the log of Github.

To show some examples of clean code we take a closer look at Bidding.py

- All methods and variables have a meaningful name there, which describes what the method/variable is for.
- All methods are as short as possible
- Each method performs only one task
- There are no code duplications (DRY)
- Since all functions are very short and only perform one task, they are very easy to read and understand. (KISS)

The Clean Code Cheat Sheet can be found in the Root Folder. Its the PDF with the name CleanCodeCheatSheet.pdf.

Continous Delivery

Travis-CI was used as continuous delivery tool. First a Python container is started with version 3.7. After that all required libraries are loaded using pip and the requierements.txt. Then all tests are executed and the results of the tests are sent to SonarCloud and Codacy. After that the metrics can be calculated and the code can be analyzed. The exact pipline can be seen in the .travis.yml in the Root Folder.

Domain Specific Language

There were 2 cases of DSL in the code. The first is in line 43 of Round.py. This initializes a Stich object, plays the Stich, assigns the cards to the winner and finally returns the winner.

```
self.turn = Stich(self.players, self.turn, self.settings).play_stich().assign_stich_to_winne
```

The second case can be found in line 29 of main.py. Here a round is initialized first, then played and finally finished.

```
self.round.setup().play_round().end_round()
```

Functional Programming

Only Final Datastructures:

- SettingContainer is only created once in main.py and then only passed on, but not overwritten/changed.

- Card objects are only created once when the cards are dealt and then passed on. A change of the cards does not take place here either. Each new round the old cards are deleted and new ones are created. This is done in the function create_shuffled_cards in Cards.py.

(mostly) side effect free functions:

- In this project we tried that all functions are as good as SiedeEffect free. This can be seen particularly well in the Bidding.py. Often the only exceptions are the functions which have to display information for the user on the terminal.

the use of higher order functions:

- The use of higher order functions can be found in several classes. Most often it occurs in the SettingContainer.py in the form of the @staticmethod Decorator. A decorator in Python is a wrapper function which takes the function written below as input and executes it. So every decorator is always a higher order function.

use clojures / anonymous functions and function as parameter:

- The function get_sorted_suit_list in the SettingContainer.py uses a lambda function, which is a parameter of the sorted function, to specify the value to be sorted by.

Code API List

Here can you find all API-References for the Classes which were used in this Project.

Bidding Functions

This File Contains the Bidding Class. Its neccessary to play the Bidding Phase

class `modules.Bidding.`**`Bidding`**`(settings, players)`
The Bidding Class contains all Functions relevant for Bidding.

`end_bidding`()
Ends the bidding Phase, if there is an bid Player, he won the bidding, else it starts new with new cards

> **Returns:** Tuple with the the player who has the next turn and the new gamstate (turn, gamestate)
> **Return type:** tuple

`get_new_turn`()
Get the new turn for the next round of bidding

`is_bidding_over`()
Checks if the bidding finished

> **Returns:** True if the bidding is finished, else False
> **Return type:** bool

`make_bid`()
Main Function for bidding. Ask Player if he wants to bid, checks if bidding Phase ends and select the new turn

`play_bidding`()
This function is the main Function of this Class. With this the Bidding Phase can be Played

> **Returns:** The Player who won Bidding or None if all passed and the new Gamestate
> **Return type:** tuple

Card Functions

This File Contains the Card Class. Each Card Object is a single Card which can be Played

class `modules.Card.`**`Card`** `(value, suit)`
   Creates a card object. This has a suit as string, as Unicode character and the value of the suit. It also includes the value of the card (7-10, J, Q, K ,A) and the points of the card value.

Returns:	A Card Object
Return type:	card

`equal_suit` `(other_card)`
   Checks if the Card has the same suit as another Card

Parameters:	**other_card** (*Card*) – Card to check if equal suit
Raises:	**TypeError** – if other_card is not of Type Card
Returns:	True if the Cards have the same suit
Return type:	bool

`get_ascii_card` `()`
   Returns the Card as Ascii Art, where each element of the list is one line of the picture.

Returns:	list with the elements of the Ascii Art Card
Return type:	list

`get_card_tuple` `()`
   Get the suit value and the card value

Returns:	tuple with suit value and card value
Return type:	tuple

`has_higher_suit_val` `(other_card)`
   Checks if the card is higher than other_card

Parameters:	**other_card** (*Card*) – Card to check with
Returns:	True if the main Card (self) is higher, False if other_card is higher
Return type:	bool

`has_higher_value` `(other_card)`
   Checks if the main card (self) has an higher value than other_card

Parameters:	**other_card** (*Card*) – Card to check with
Returns:	True if the main card (self) is higher
Return type:	bool

`ishigher` `(other_card, check_suit_val=False)`
   Checks if the card is higher than other_card

Parameters:	**other_card** (*Card*) – Card to check with **trumpf** (*str*) – Trumpf which is played at the moment **order_dict** (*dict*) – dictionary with the ranking order of the cards **check_suit_val** (*bool, optional*) – If the higher suit wins, if no card is Trumpf and there are not equal suit. If False the main card (self) is winning. Defaults to False.
Raises:	**TypeError** – if other Card is not of Type Card
Returns:	True if the main Card (self) is higher, False if other_card is higher
Return type:	bool

istrumpf **()**
  Checks if the Card is Trumpf

Parameters:	**trumpf** (*str*) – the trumpf which is played at the moment
Raises:	**TypeError** – if trumpf not None or not string
Returns:	True if card is trumpf
Return type:	bool

print **()**
  Prints the Ascii Art Card

same_suit_or_trumpf **(**other_card**)**
  Check if 2 Cards have the same Suit or both are trumpf. :returns: True If the 2 Cards have the same suit or are both trumpf, else False :rtype: boolean

class modules.Card.**EmptyCard**
  Creates an Empty card, with no text

Cards Functions

This File contains the Cards object. This is a container CLass for Multible Card Objects

class modules.Cards.**Cards** (settingContainer, cards=None)
  Creates a Cards object, which is a container for multible Card objects. It has functions to manipulate multible Card objects.

add_card **(**card**)**
  Add a new Card to the Cards object

Parameters:	**card** (*Card*) – The Card that should be added

add_card_and_sort **(**card**)**
  Add a new Card to the List and sort the List afterwards

Parameters:	**card** (*Card*) – The new Card the should be Added

create_shuffled_cards **()**
  Creates 32 cards and returns them shuffled

empty_cards **()**
  Remove all Cards from the Cards Object

get_jacks **()**
  Get all Jacks from the Cards Object

Returns:	A List with all Jack Card objects
Return type:	List

index **(**card**)**
  Reeturns the Index of the Card in the list of all Cards

Parameters:	**card** (*Card*) – The Card, where the Index should be found
Returns:	The index of the Card in the Cards Object list of all Cards
Return type:	int

print_cards_ascii **(**card_delimiter=':'**)**
  Prints all Card objects as Ascii-Art in the Terminal.

Parameters:	**card_delimiter** (*str, optional*) – The Symbols which are used between 2 Cards. Defaults to ":".

remove (`card_to_remove`**)**
   Remove a Card from the List

>   **Parameters:**   **card_to_remove** (*Card*) – The Card the should be removed

sort_cards ()
   Sorts all Card objects with a simple Bubble sort. There can only be 12 Cards at your Hand, so Performance isnt that important

Player Functions

This File Contains the Player Class which has all Informations about a single Player

class `modules.Player.`**`Player`** (`player_name, **settings`)
   This Class contains all Infomations about a Player

has_cards ()
   Checks if the Player has Cards

>   **Returns:**   True if the Player has Cards, else False
>   **Return type:**   bool

Players Functions

This File contains the Player CLass. which is a container for Player objects

class `modules.Players.`**`Players`** (`settingContainer, **kwargs`)
   The Player Class is an container for multible Player Objects

get_next_player (`player`**)**
   Get the next Player in the List, if the list is at the End, it starts from the beginning

>   **Parameters:**   **player** (*Player*) – The Player at the moment
>   **Returns:**   the Player, after the Player which was given as input
>   **Return type:**   Player

get_player_by_num (`num`**)**
   Get a Player by its num which was assigned when creating a Player object

>   **Parameters:**   **num** (*Int*) – The num of the Player
>   **Raises:**   **Exception** – If a Player with this num doesen't exist
>   **Returns:**   The Player with the corresponding num
>   **Return type:**   Player

reset ()
   Reset the Player for a new Round.

set_players_on_next_position ()
   Change the positions of all Players. Forhand –> Middlehand, Middlehand –> Backhand, Backhand –> Forhand

sort_cards ()
   Sort all Cards of all Players

Round Functions

The File with the Round Class

class `modules.Round.`**`Round`** (`players, settingContainer`)
   The Round Class has all Functions for Playing and Evaluating a Played Round

end_round ()
    Calculates all Points and ends the played round

        Returns: returns itself
        Return type: Round

give_cards ()
    give Cards to every Player and put 2 into the Skat

play_round ()
    Play a round

        Returns: return its own object
        Return type: Rround

set_card_default_values ()
    sets the static Values of Card to its default

set_gamemode ()
    Ask the SIngle Player which play Types he wants to chose and sets it

setup ()
    Starts the Setup, in which the single Player can take the Skat and choose the gamemode

        Returns: returns itself
        Return type: Round

start_bidding ()
    Start and play the bidding phase

        Returns: the new gamstate after the bidding Phase
        Return type: int

start_new_round ()
    Starts a new round

SettingContainer Functions

This File contains the SettingContainer Class

class modules.SettingContainer.**SettingContainer** (setting_dict)
    This Class contains all Settings which are relevant for the Game.

    static **create_SettingContainer_from_file** (language='en')
        creates a SettingContainer Object from a setting.json file

            Parameters: **language** (*str, optional*) – the Language of the game. Defaults to "en".
            Returns: returns a SettingContainer Object
            Return type: SettingContainer

    get_sorted_suit_list ()
        returns a sorted list of all suits

            Returns: the List of string with the suits
            Return type: List

Stich Functions

This File Contains the Stich Class

class `modules.Stich.`**`Stich`**`(players, turn, settings)`
   The Stich CLass contains all functions which are neccessary to play a single Stich

   `assign_stich_to_winner` **()**
      assign the Cards of the Stich to the winner

Returns:	retunrs itself
Return type:	Stich

   `get_winner` **()**
      Gets the cards that were Played and returns the number of the Player who won

   `play_stich` **()**
      Play a single Stich

Returns:	returns itself
Return type:	Stich

   `print_card_table` **()**
      Prints the Cards that were Played plus empty cards to show how many need to be played

Tools Functions

Here are functions witch are needed in diffrent states of this programm

`modules.tools.`**`get_user_card`**`(show_message, error_message, user_cards)`
   Get a Card from a Userinput

`modules.tools.`**`get_user_true_false`**`(show_message, error_message, cards)`
   Get True or False if user wants to take the bid

`modules.tools.`**`user_select_card`**`(show_message, error_message, user_cards)`
   Letthe User select a Card from ts Cards

Parameters:	• **show_message** (*Str*) – Message whitch will be shown befor selecting
	• **error_message** (*str*) – Message that will be shown if an error occures
	• **user_cards** (*Cards*) – the Cards of the user
Returns:	returns the new User Cards without the selected card and the selected card
Return type:	Tuple

IDE

Visual Studio Code was used as IDE. This is a very lightweight IDE which can easily be extended with packages. The most important shortcuts are F2 to rename and customize the names of functions and variables throughout the code and Shift+Alt+F to format the code.

Metrics

In this project 2 different software solutions were used to create different metrics.

The first tool used was **SonarCloud**, which is the cloud solution of **SonarQube**. The dashboard can be found at https://sonarcloud.io/dashboard?id=PascalStehling_Skat Metrics that were especially important is:

   • *Duplications*: Percentage of all rows, how many of them are duplicates

   • *Coverage*: percentage of how many lines of code are covered by tests.

Other metrics are also created by **SonarQube**, such as *security*, *reliability* and *maintainability*.

As second tool **Codacy** was used. This is also an online solution that can be easily linked to github accounts. The setup and usage is more user-friendly than **SonarQube**. The dashboard can be accessed via

https://app.codacy.com/manual/PascalStehling/Skat/dashboard. **Codacy** can also calculate the metrics *Duplications* and *Coverage*. But there are also other metrics like *percentage of errors in code* and *percentage of complex files*.

All in all, I think **Codacy** is better than **SonarQube** because it is easier to get started and settings can be made more easily via the website.
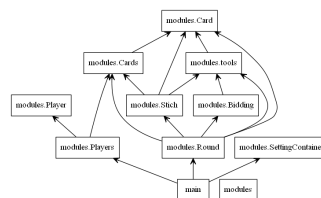
UML Diagrams

To better understand the code and the basic concept, 3 UML diagrams were created.

The first is an activity diagram, which shows the basic flow of a skat game.
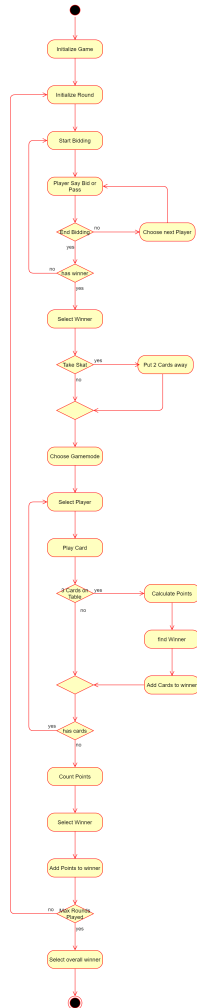
The second one is a package diagram and shows which files call each other and are connected.

The third diagram is a class diagram that shows the general structure of the classes and also some connections between them.
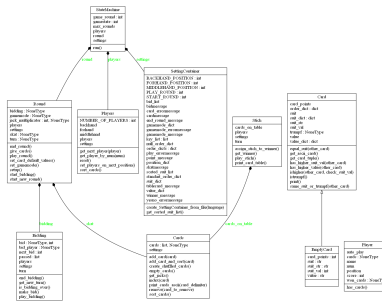
Unittests

A total of 37 unit tests were written for the classes of this project. According to Codacy, this represents a total coverage of 73%. The tests were written using the standard library Unittest from Python. The tests can be found in the tests folder and run using *bazel run test* or *doit run_test*. All unit tests can be found in the tests folder. However, there is not yet a unit test for every class.

All in all I found this project very useful, because I could try out many different things around programming. I found a lot of problems that I always made myself and new ways to improve the code. If I had to do this project again I would probably do the whole project in a test-driven approach.

Indices and tables

- **genindex**
- **modindex**
- **search**

Index

U

Python Module Index

m

modules

modules.Bidding

modules.Card

modules.Cards

modules.Player

modules.Players

modules.Round

modules.SettingContainer

modules.Stich

modules.tools