

# FRPS - Flexible Rock, Paper, Scissors

Teilnehmer: Thomas Knott

## Aufbau des Projekts:

Das Projekt beinhaltet die Implementierung von Rock, Paper, Scissors, gemäß den Projektvorgaben.  
Darüberhinaus kann das Script als Modul in andere Python-Skripte eingebunden werden.  
Hier wurde es in das Übungsscript "feline\_fantasy\_1.py" eingebunden.

## Struktur:

### Dateien und Ordner:

```
frps
├── frps_classes.py
├── frps_functions.py
├── frps.py
├── __init__.py
├── res
│   ├── rpsls.yaml
│   └── rps.yaml
```

`frps.py`:

Hauptscript. Einzeln aufgerufen startet es ein FRPS-Spiel mit den Regeln und Optionen gemäß den Mindestanforderungen für das RPS-Projekt.

Die Regeln und Optionen lassen sich leicht über YAML-Dateien oder Attribute der Game-Klasse anpassen.

Als Modul stellt es die nötigen Klassen und Funktionen bereit, um es in andere Spiele (z.B. als Battlescreen) einzubauen.

`frps_functions.py`:

Enthält allgemeine Funktionen, die von den Klassen oder dem Hauptscript verwendet werden können.

`frps_classes.py`:

Enthält die Klassen und Methoden für den Spielstatus und -ablauf.

## Benötigte Module/ Imports:

- random
- time
- yaml

## Ablauf bei Aufruf von `frps.py` als Script:

1. Die Spielerin wird gefragt, welchen Regelsatz er verwenden möchte (klassisches RPS oder Rock, Paper, Scissors, Lizard, Spock).
2. Die Spielerin legt fest, nach vielen Siegen das Spiel als gewonnen gilt.
3. Es wird ein "Game"-Objekt erzeugt und eine while-Schleife gestartet. Diese endet, wenn die Zielpunktzahl erreicht ist.
4. Die Spielerin gibt seine gewünschte Aktion (z.B. "Rock") ein.
5. Die Computerspielerin entscheidet ihre Aktion zufällig.
6. Die jeweiligen Entscheidungen werden ausgegeben und das Ergebnis der Runde berechnet.
7. Das Ergebnis der Runde wird ausgegeben.
8. Falls die Zielpunktzahl noch nicht erreicht ist wird eine neue Runde gestartet.

## Anmerkungen zur Implementation:

Das Script / Module ist darauf angelegt möglichst flexibel zu sein. Es wird ein verschachteltes Python-Dictionary benutzt, um die Regeln des Spiels kompakt abzubilden. Da die Regeln aus einer YAML-Datei gelesen werden, ist es sehr einfach eigene Varianten von FRPS zu erstellen oder es in anderen Projekten einzubinden.

## Beispiel: Klassisches RPS

### YAML:

```
Rock:
  Scissors: crushes
Paper:
  Rock: covers
Scissors:
  Paper: cut
```

- Rock schlägt Scissors und nutzt dabei das Verb "crushes".
- Paper schlägt Rock und nutzt dabei das Verb "covers".
- Scissors schlägt Paper und nutzt dabei das Verb "cut".

### Resultierendes Regelset als Python-Dictionary:

```
ruleset_classic = {
    "Rock": {"Scissors": "crushes"},
    "Paper": {"Rock": "covers"},
    "Scissors": {"Paper": "cut"}
}
```

## Beispiel: Ablauf

Usereingaben sind grün gefärbt.

```
Which Game do you want to play?
[1] Classic Rock, Paper, Scissors
[2] Rock, Paper, Scissors, Lizard, Spock
Input: 1
How man victories are needed to win the game?
Input: 1

Round number: 1
Choose attack: [Rock], [Paper], [Scissors]: Rock
Player chooses: Rock
Computer chooses: Paper
    Paper covers Rock
    You lost this round.
Points left to win the game: 1

Round number: 2
Choose attack: [Rock], [Paper], [Scissors]: Paper
Player chooses: Paper
Computer chooses: Rock
    Paper covers Rock
    You won this round!
Points left to win the game: 0
Gametime: 7s

***YOU ARE WINNER***
```

## Als Modul:

FRPS lässt sich als Modul importieren.

```
import frps
```

Danach stehen die Klassen und Funktionen zur Verfügung.

Zunächst muss eine Instanz von "Game" erstellt werden, in der die allgemeinen Spielparameter (insb. die zu verwendende YAML-Datei) festgelegt werden.

```
my_frps =
frps.Game(ruleset=frps.load_ruleset_from_file("ffl_rps.yaml"),
rounds=1, target_score=1)
```

Zu dem Zeitpunkt, an dem die Spielerin eine Aktion wählt müssen dann die Choice-Objekte definiert werden.

```
my_player_turn = frps.Choice(my_frps, choice=my_player_input)
```

```
my_ai_turn = frps.Choice(my_frps,  
choice=random.choice(list(my_frps.ruleset.keys()))))
```

Dann muss die Spiellogik analog zu der in `frps.py` (oder auch anders, Experimente sind möglich) angelegt werden.

Eine Beispielimplementation ist in `feline_fantasy_I.py` umgesetzt.

## Kurzbeschreibung Feline Fantasy I:

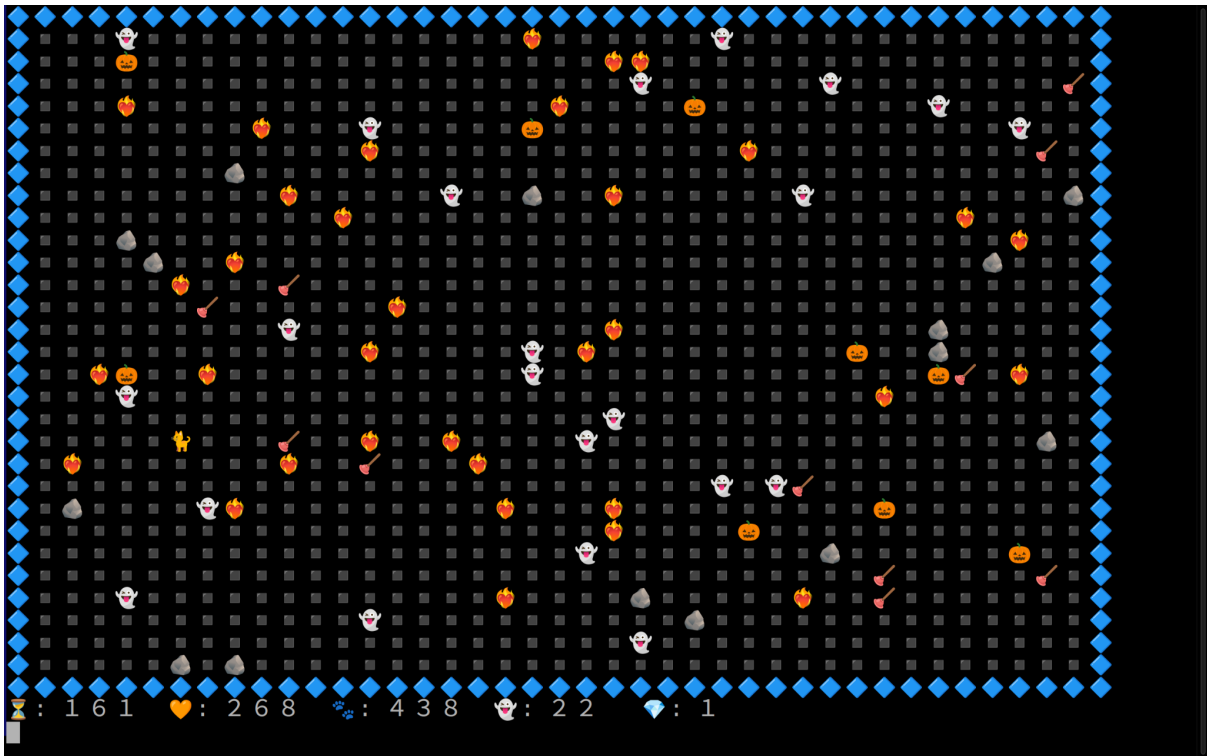
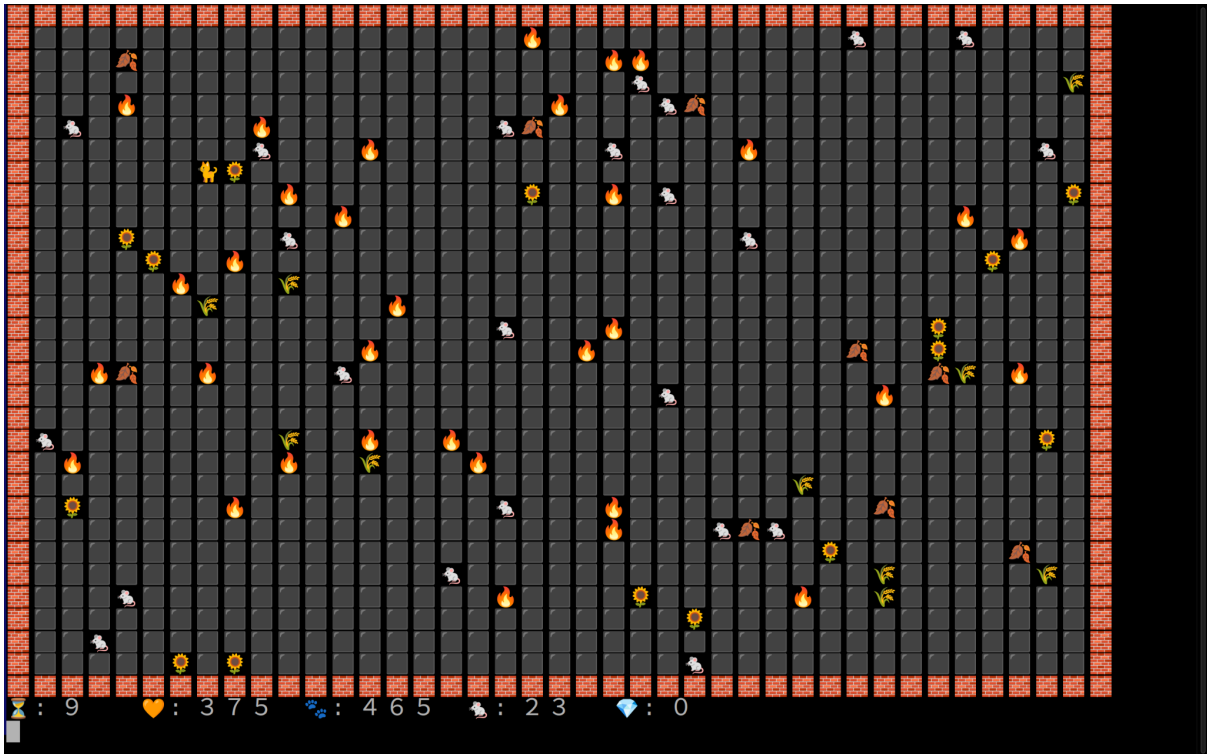
In Feline Fantasy I schlüpft die Spielerin in die Rolle einer Katze. Das Ziel ist es eine gewisse Anzahl von Mäuse zu fangen ohne zu sterben.


Die Katze läuft selbstständig, kann aber jederzeit auch über die Pfeiltasten bewegt werden. Das manuelle bewegen kostet jeweils einen Schritt. Auf dem Spielfeld befinden sich diverse Pflanzen, Mäuse und Feuer. Pflanzen haben keine Auswirkung und dienen nur als Dekoration. Berührt die Spielerin ein Feuer verliert sie Lebenspunkte. Berührt die Spielerin eine Maus wird ein Kampf über FRPS gestartet. Gewinnt die Spielerin den Kampf wird die Maus entfernt und die Spielerin bekommt einen Punkt, sowie einige Lebenspunkte. Verliert die Spielerin den Kampf verliert sie Lebenspunkte. Alle paar Sekunden wechselt das Spielfeld zu einem Halloween-Theme. Dies ist rein dekorativ und hat keinen sonstigen Einfluss auf das Spiel.

Das Spiel ist gewonnen, wenn alle Mäuse auf dem Spielfeld gefangen wurden.


## Screenshots:

Die tatsächliche Darstellung kann abweichen, da nach Erstellung dieser Dokumentation noch Verbesserungen eingebaut wurden. Der aktuelle Code kann auf Anfrage ([thmskntt@gmail.com](mailto:thmskntt@gmail.com)) über GitHub heruntergeladen werden.





```
BATTLE STARTED! Press ENTER to begin
```



```
Cat chooses: speed  
Mouse chooses: physical  
                speed outruns physical  
YOU WON!  
BATTLE ENDED. Press ENTER to go back.█
```

#### Disclaimer:

`feline_fantasy_I.py` ist das Resultat aus den diversen Übungen des Kurses (insb. "Dungeon Crawler). Der Code ist nicht optimiert und ziemlich chaotisch.

Es wurde nur lokal unter Linux (aktuelles Ubuntu, KDE Desktop, Konsole als Terminal) getestet.

Das Script läuft nur sauber in einem "echten" Terminal (Konsole, xterm, etc. Eventuell Putty/Kitty unter Windows) und nicht in der Python oder PyCharm-Konsole.

Desweiteren muss das Terminal mit einer UTF-8 Schriftart laufen, die auch die full-width Zeichen (insb. Emoticons / Emojis) beinhaltet. Empfohlen und getestet ist dafür die open source Font "Noto Color Emoji": <https://fonts.google.com/noto/specimen/Noto+Color+Emoji>

Es werden folgende externen Pakete benötigt:

- numpy
- pynput
- pydub

Das Kampfsystem ist noch nicht fertig implementiert. Die gezeigten Optionen (physical, speed, scare) sind eigentlich nur Typen, denen die eigentlichen Aktionen zugeordnet werden.

Beispiel:

Cat chooses: Bite

Mouse chooses: Run away

"Bite" ist Typ "physical" und "Run away" ist Typ "speed". Im Regelfile ist definiert, dass "speed" gegen "physical" gewinnt.

## Probleme und sonstige Anmerkungen:

- Grafik über Emojis in einem Terminal zu machen war eine bewusst dumme Idee zu Übungszwecken. Eigentlich wäre pygame sinnvoll.
- Musik in feline\_fantasy\_1.py ist eingebaut, aber die Library (pydub) flutet den stderr-Kanal mit Output von ffmpeg. Das ist nervig und verhindert die Textausgabe von FRPS. Die Musik wurde daher deaktiviert. Die verwendete bzw. nicht-verwendete Musik stammt von <https://www.fesliyanstudios.com/royalty-free-music/download/a-bit-of-hope/565>

**Todo:** Bessere Sound-Library finden.