Prosjekt Rapport

ITF10219-1 20H
Programmering1

Kandidatnr: 192793

Østfold University College
Faculty of Computer Sciences

Innhold

Introduksjon:

Planen min var å lage en Casino-spill kalt Baccarat, men det ble for vanskelig og valgte å lage Flash Card spill. Problemet med Casino spill var at den krevde mye design og spesifikk kunnskap rundt

User Interface. Jeg valgte flash Card fordi her bruker jeg ganske masse tid på logikk i koden enn design.

Spillet går ut på at brukeren skal gjette navn på kjendiser. Brukeren har fire valg i menyen; 1. spill spillet 2. vise log 3. vise regler 4. avlutt spill

Videre skal jeg forklarer dypere funksjoner som er brukt og valg som spilleren har.

Valg i menyen:

Hvert valg i menyen vil vise en label i **tkinter** derfor trenger jeg en funksjon for å fjerne gamle label før neste label skal vises. Denne funksjonen heter **hide_all_frames** som har en forloop gjennom **winfo_children()** som returnerer list av alle barn til en widget og sletter alle med **.destroy()** funksjonen.

1. Spill

Når brukeren trykker på valget 'play' kjøres funksjonen **show_celebrities ().**
Denne funksjonen vil vise et tilfeldig bilde av kjendiser. Deretter vil den pakke(.pack) opp en **Entry**, to **knapper** og en **label**.

2. Vis log:

Når brukeren trykker på valget 'show log' kjøres funksjonen **show_log ().** Da etter å ha slettet forrige frames, vil en **Text ()** bli pakket i egen label. Deretter åpner programmet en fil(log.txt) som inneholder alle bruker log. Før den skal vise innholdet på log.txt, slettes gamle innhold. Her en **button** blir packed også, den lar brukeren fortsette å spille uten å starte ny runde. Dette er veldig nyttig synes jeg.

3. Vis regler:

Når brukeren trykker på 'show log' kjøres funksjonen '**show_rules'**. Da etter å ha slettet forrige frames, vil en frame bli pakket i egen label. Programmet åpner 'rules.txt' og leser innholdet på filen. Jeg forklarer ikke om regler her, dere kan lese dem i menyen i spillet.

4. Avslutt spill:

Hvis brukeren trykker på valget' Exit' vil 'command = root.quit' kjøres og spillet er over.

Funksjoner:

Videre i koden bruker jeg **global** statement ganske mye i alle funksjoner.
Problemet er at siden jeg ikke kan OOP, men vil at endret verdi for en variabel være samme for alle funksjoner i **global Scope**. Jeg vet at det er ikke beste praksis å bruke **'global'** så mye.

1. show_celebrities:

Etter å slette forrige frame vil den funksjonen definere en random celebrity_Image. For dette brukes **randint** fra **random library**. Etterpå ved hjelp av **PIL**(Pillow) library vil bli packed i egen frame.

Deretter vil en '**Entry'**, to knapper og en '**label'** under de knapper bli pakka i samme frame. Altså 'celebrities_frame' blir laget og det er 'interface' i spillet.

2. check_answer:

Endre answer form:

Denne funksjonen vil først endre bruker-input form, slik at den fjerner **mellomrom** (' ') og erstatter store bokstaver med små bokstaver.
Videre sjekker funksjonen om det finnes tall eller symbol som er **ikke alfabetisk**, da vil bruker få en melding om å bruke kun bokstaver. Dersom brukeren ikke skriver noe og trykker på 'answer' knappe, vil dem få melding om det.

Hvis svaret er riktig og brukeren har fortsatt forsøk igjen fra 10 forsøk,
Vil de få melding 'correct'. Utfordring var at siden alle bilder dukker opp random, brukeren kan få poeng for duplikat svar. Dette ordnet jeg ved å legge til en **if** statement som sjekker den svaret er riktig og finnes allerede i log.
Log er en liste som inneholder **Dictionary(entry_log).** Entry_log ser sånn ut:
{'attempt': 1, 'input': 'kyliejenner', 'response': 'Correct', 'score': 10, 'time': '22:47:21'}

For å finne duplikat svar fra log, bruker jeg **List Comprehensions** som returnerer list.

Finne total_score:

Det neste trinnet er å finne total score i hvert tidspunkt, her brukte jeg 3 forskjellige metoder:

1. Brukte en **list** som inneholder alle riktig svar som brukeren har oppgitt. Den lista blir til et **set-object.** Deretter en forloop gjennom **set_object** vil appende antall ganger svar som har blitt skrevet mer enn en gang.

```
total_appearances = []
temp = set(right_answers)  # temp : {'kyliejenner': 3, 'howardstern': 2}
for i in temp:
    number_of_appearances = right_answers.count(i)
    total_appearances.append(number_of_appearances-1)  # total_appearances = [2,1]

index = sum(total_appearances) # index = 3
```

I koden ovenfor finner jeg index som brukes i;
new_sum = old_sum (index * 10)

2. Den andre metoden jeg lærte som var kul, er å bruke **len ()** på set-object av liste som inneholder riktig svar skrev av brukeren.

   temp = set (right_answers)
   index = len (right_answers) – len (temp)

3. Den siste er enklest, ved bruk av **list Comprehensions**. Her endres score fra 10 til 0 hvis svaret er riktig og det finnes allerede i log (liste med entry_log).
   Etter å ha endret kode logikken 3 ganger brukte jeg den siste metoden **med list Comprehensions** i koden.

Etter å ha kalkulert totale_score i forrige delen vil programmet skrive log (liste med entry_log) i en json file.

Main.py

Jeg tenker å forklare tre ting i main.py som ikke ble sagt tidligere i rapporten.

1. Øverst i main.py etter å ha definert en størrelse (900 * 700) for grensesnittet vårt, lages det en label som heter status_bar. Text på den status_bar er ‘No error caught’.
   Denne label brukes for å vise eventuelle error i programmet. Senere i rapporten forteller jeg mer om error håndtering i prosjektet.

2. Definerer en **path**(‘celebrity_name.csv’) sammen med en **liste**(‘data’). Her kalles funksjonen **read_from_csv** fra andre moduler i prosjektet og brukes videre for å vise bildet av en kjendis i **show_celebrities** ()

3. Alle globale variabler er skrevet i egen .py(all_variables) og hentes i Main.py

Error håndtering:

Index-error:

Oppstår ofte når vi skal bearbeide list særlig hvor vi iterer gjennom for lopp. Jeg ville implementere try-except i ***read_from_csv_file.py***. Der finnes det en funksjon (read_from_csv) som har forloop og itererer gjennom en liste. Problemet var at **status-bar** som skal vise mulige error etter error handling er en label definert kun i **Main**. Derfor kunne ikke bruke status_bar for å skrive mulige error men brukte try-catch i selve funksjonen.

FileNotFoundError:

Oppstår ofte når man oppretter filer eller Dictionary. En annen tilfelle kan være når man leser fra eller skriver til fil.

Funksjoner som kan gi oss den typen error er **read_from_csv ()** og **show_log**(). Funksjoner importeres fra andre moduler og siden jeg **har ikke tilgang til stataus_bar fra andre moduler**, implementerte jeg try-catch når de funksjoner **blir kalt tilbake i main.py**. Hvis feil oppstår, vil text på status bar vise feilen.

NameError:

Oppstår når en global eller lokal variabel finnes ikke, i funksjonen **write_to_json (file, my_dict)** Hvis navn på **my_dict** er feil eller eksisteres ikke vil jeg få en NameError. Bruker try-catch i main.py når programmet skal skrive log til Json-fil.

Finally:

I mitt prosjekt hadde jeg ikke behov for **finally** etter try-**catch**, siden man får vite hvordan error har oppstått uten at programmet krasjer helt.

Hva var vanskelig?

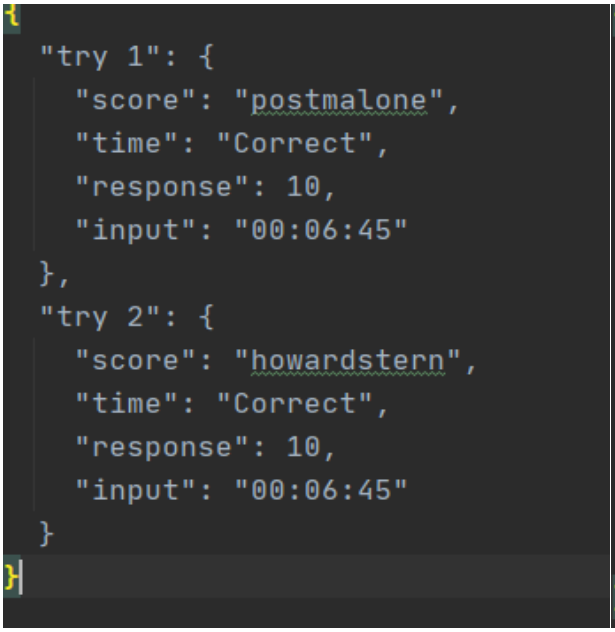
```
"""  parameters : keys = ['input1', 'response', 'total_score', 'time'] which is defined by me
                  input_log, response_log, time_log and score_log is generated when user presses the answer button
                  attempt_x is a string and x is the try number which is generated when user presses the answer button
                  values is a empty list that will be explained later i description
    description :
    1. take the last values of each list and append them to new list called values
    2. make a dictionary with keys and values called dictionary1
    3. make a nested dictionary with using attempt_x as key and dictionary1 as value
    """
def change_data_to_nested_dictionary(keys, values, input_log, response_log, time_log, score_log, attempt_x):
    values.append(input_log[-1])
    values.append(response_log[-1])
    values.append(time_log[-1])
    values.append(score_log[-1])

    dictioanary1 = dict(zip(keys,values))
    # dictioanary1 = {'input': 'Bob', 'response': 'correct' , 'total score ': '20' , 'time' : '21:00' }
    final_dict = {attempt_x: dictioanary1}
    # final_dict = try1': {'input': 'Bob', 'response': 'Mgr' , 'total score ': '100' , 'time' : '21:00' }
    return final_dict
```

Til slutt har tar jeg med en funksjon som var vanskelig. Bildet ovenfor er funksjonen change_data_to_nested_dictionary () som brukes for å gjøre om **fire list** til en **nested-dictionary.** Alle parametere er forklart i koden. Videre forteller jeg et eksempel ved bruk av denne funksjonen.

```
# TEST
first = [1, 2, 3, 22]
second = [4, 5, 6, 12]
third = [7, 8, 9, 32]
forth = [10, 11, 12, 42, 33]
values = []
keys = ['input', 'response', 'score', 'time']
final_dict = {}
attempt_x = 'attempt1'
my_dict =change_data_to_nested_dictionary(keys, values, first, second, third, forth, attempt_x)
# my_dict = {'attempt1': {'input1': 22, 'response': 12, 'total_score': 32, 'time': 33}}
```

I bildet øverst er det fire lister som skal bli til en nested_dictionary. Jeg har kjørt den koden og alt fungerer som skal, men i det neste bildet viser resultatet når jeg implementerte denne funksjonen i prosjektet mitt.



Som dere ser noe skikkelig rart skjer her, **Keys** for **Value** vil ikke bli tildelt verdi i samme rekkefølge hver gang. Bildet til høyre er riktig oppsett i log, men hver gang jeg trykker på run, vil det bli laget en tilfeldig

Key: Value. Jeg brukte mye tid på dette men til slutt kom jeg til konklusjon at **zip()** vil sette opp tilfeldig **key-value** par i koden min , men ikke i test-koden.

Det som er brukt i min kode nå, er en **hardkoda Dictionary** som lages for hvert forsøk i spillet, Slik

log entry = {"attempt": tries, "input": answer, "response": response, "score": score, "time": current_time}