

TDT4113 - Datateknologi, programmeringsprosjekt

Oppgave 1: Stein, Saks, Papir

Dette dokumentet beskriver den første oppgaven i ProgLab 2, som vi kaller “Stein, Saks, Papir”. For denne oppgaven gjelder at:

- Oppgaven skal løses individuelt
- Oppgaven skal løses med objektorientert kode skrevet i python
- Fristen for oppgaven er *2 uker*, dvs. implementasjonen din demonstreres senest 2. September 2015 kl 14:00.

1 Om spillet “Stein, Saks, Papir”

I den versjonen av spillet vi skal se på er det 2 spillere, Spiller 1 og Spiller 2. Hver spiller kan velge en av tre aksjoner: “**stein**”, “**saks**” eller “**papir**”. I spillets første fase gjør spillerne sine valg uten å avsløre dem for hverandre. I andre fase viser de to spillerne samtidig hva de har valgt, og det bestemmes en vinner. Dersom Spiller 1 og Spiller 2 har valgt det samme er det uavgjort, hvis ikke kåres vinneren ut fra følgende regler (se også Figur 1):

- **stein** slår **saks**.
- **saks** slår **papir**.
- **papir** slår **stein**.

2 Implementasjon

Implementasjonen du lager skal skrives i objektorientert Python. For en rask oppfriskning av hvordan man tenker objektorientert, samt litt om hvordan vi gjør

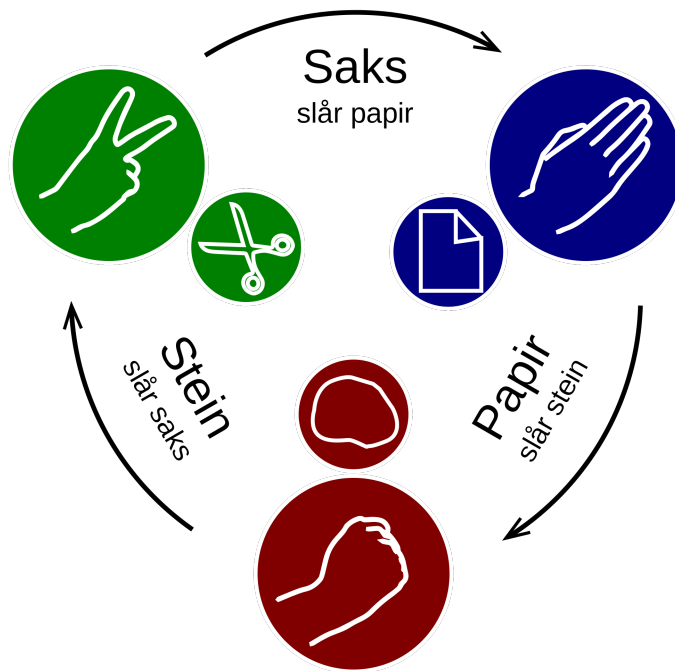


Figure 1: Hvordan bestemme vinner i Stein, Saks, Papir.

dette i Python kan du sjekke innføringsvideoen som ligger på fagets wiki-sider. Det står også litt om dette i kapittel 7 i boka (“Exploitations in Computing”).

Vi skal implementere en del halv-smarte spillere til “Stein, Saks, Papir” og to test-omgivelser for å prøve dem ut. Det kan også være nyttig å utvikle noen hjelpe-klasser for å gjøre implementasjonen så enkel som mulig. Vi diskuterer disse i tur og orden nå.

2.1 Spillerne

Det er anbefalt å lage en klasse for spillere. Vi kaller den `Spiller` her. `Spiller` bør (minst) ha følgende metoder:

- `velg_aksjon`: Denne metoden velger hvilken aksjon som skal utføres (spille `stein`, `saks` eller `papir`) og returnerer dette.
- `motta_resultat`: Etter at et enkelt-spill er over får spilleren vite hva som ble valgt av begge spillere samt hvem som vant. Den kan velge å lære fra denne informasjonen (se spesielt spillerne `Historiker` og `MestVanlig` lenger ned i beskrivelsen).

- `oppgi.navn`: Denne metoden oppgir navnet på klassen, slik at vi kan rapportere dette i grensesnittet.

Vi skal lage et sett av spillere, og et minstekrav er å er gitt i denne listen:

- **Tilfeldig**: Denne velger tilfeldig om den skal gjøre **stein**, **saks**, eller **papir**. For å implementere denne kan det for eksempel være greit å bruke `random.randint(0, 2)`.
- **Sekvensiell**: Denne spilleren går sekvensielt gjennom de forskjellige aksjonene, og spiller **stein**, **saks**, **papir**, **stein**, **saks**, **papir**, **stein**, **saks** ... i en fast sekvens uansett hvordan motstander oppfører seg.
- **MestVanlig**: Denne spilleren ser på motstanderens valg over tid, og teller opp hvor mange ganger motstander har spilt **stein**, **saks** og **papir**. Så antar den at motstander igjen vil spille det som han/hun/den har spilt mest av så langt, og **MestVanlig** velger derfor optimalt ut fra dette. I det første spillet, der **MestVanlig** ikke har sett noen av valgene til motstander og dermed ikke vet hva den bør velge, velger den i stedet tilfeldig.

Eksempel: La oss anta at motstander har spilt denne sekvensen: **stein**, **saks**, **stein**, **stein**, **papir**, **saks**, **papir**, **stein**, **papir**, **stein**, **stein**, **saks**, **papir**, **saks**. Vi ser at **stein** er mest vanlig i historien til motstander, så **MestVanlig** antar at **stein** vil komme igjen. Trekket fra **MestVanlig** er følgelig **papir** (ettersom **papir** slår **stein**).

- **Historiker**: Denne spilleren ser etter *mønstre* i måten motstander spiller på. **Historiker** defineres med en parameter “**husk**”. Beskrivelsen starter med å se på situasjonen **husk=1**.

Eksempel: La oss igjen anta at motstander har spilt sekvensen **stein**, **saks**, **stein**, **stein**, **papir**, **saks**, **papir**, **stein**, **papir**, **stein**, **stein**, **saks**, **papir**, **saks**. **Historiker** ser på det siste valget (**saks**), og går tilbake i historien for å finne hva motstander pleier å spille *etter* en **saks**. Motstander har spilt **saks** tre ganger tidligere; to av disse ble etterfulgt av **papir**, mens en ble etterfulgt av **stein**. **Historiker** tenker dermed at det mest vanlige etter en **saks** er **papir**, og antar derfor at den neste aksjonen til motstander blir **papir**. **Historiker** velger dermed **saks** (fordi **saks** vinner over **papir**).

Når **husk** er større enn 1 leter **Historiker** etter sub-sekvensen bestående av de **husk** siste aksjonene i stedet for kun å se etter den aller siste når den skal bestemme seg for hva den skal spille. Dersom **husk=2** betyr det at **Historiker** først sjekker hvilke 2 aksjoner som ble spilt sist. I dette tilfellet var det (**papir**, **saks**), og **Historiker** vil derfor lete etter denne sub-sekvensen

i historien. Kombinasjonen er bare spilt en gang tidligere i den rekkefølgen, og da valgte motstanderen å fortsette med **papir**. **Historiker(husk=2)** antar dermed at det også nå blir **papir** fra motstander etter (**papir, saks**), og velger følgelig selv **saks**.

Hvis vi ser på den samme sekvensen men har **husk=3** må vi lete etter sub-sekvensen (**saks, papir, saks**) i historien. Denne sekvensen er ikke spilt tidligere, og **Historiker** skal i slike tilfeller velge tilfeldig.

2.2 Hjelpesklasser

Det kan være nyttig å definere en **Aksjon**-klasse, som representerer aksjonen som velges (“Spille **stein**”, “Spille **saks**”, eller “Spille **papir**”). Poenget med denne klassen er å definere om en **aksjon** vinner over en annen. Dette gjøres ved å definere `__eq__` og `__gt__`; se introduksjonsvideoen om objektorientert Python, spesielt om operator overloading.

Klassen **EnkeltSpill** kan brukes til å gjennomføre og representere et enkelt spill. **EnkeltSpill** brukes til å spørre etter spillernes valg, finne hvem som har vunnet, rapportere tilbake til spillerne samt rapportere til konsollet. Klassen trenger i såfall disse metodene:

- `__init__(self, spiller1, spiller2)`: Initiere en instans av klassen, der **spiller1** og **spiller2** er de to spillerne.
- `gjennomfoer_spill`: Spør hver spiller om deres valg. Bestem resultatet ut fra regelen at det gis ett poeng til vinneren og null poeng til taperen (ved uavgjort får begge spillere et halvt poeng). Rapport valgene og resultatene tilbake til spillerne.
- `__str__`: Tekstlig rapportering av enkelt-spillet: Hva ble valgt, og hvem vant?

2.3 Test-omgivelser

For å teste spillerne våre skal dere implementere to forskjellige test-omgivelser.

2.3.1 Tekst-basert turnering for å teste to automatiske spillere

Den første test-omgivelsen, her definert i klassen **MangeSpill**, må minst ha funksjonalitet tilsvarende disse metodene:

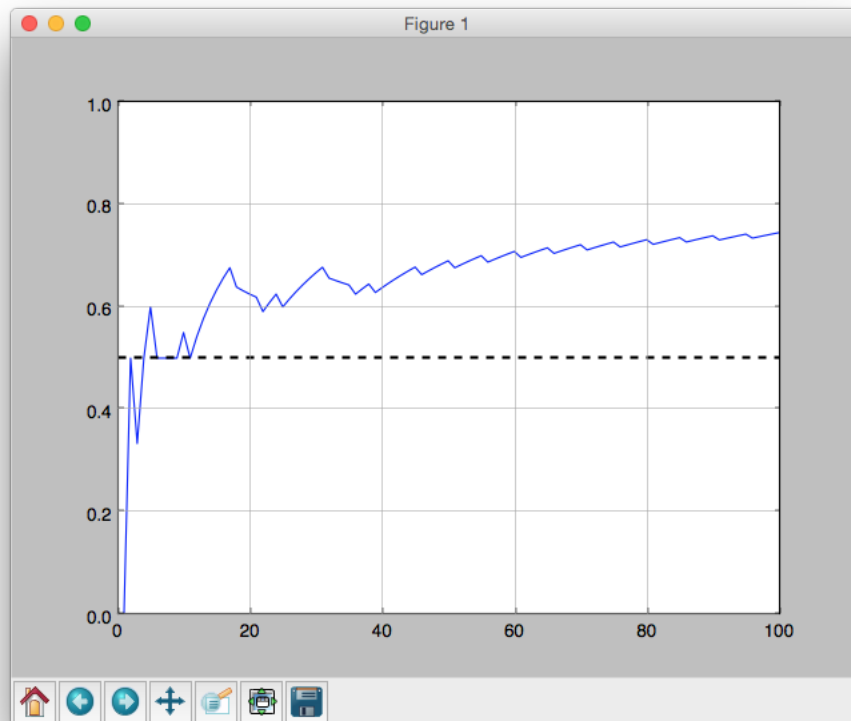


Figure 2: Utviklingen av gjennomsnittlig antall poeng per spill for Historiker(2) mot MestVanlig. Turneringen går over 100 spill og viser at Historiker(2) ganske hurtig får en god ide om hva den bør gjøre.

- `_init_(self, spiller1, spiller2, antall_spill)`: Sette opp instansen med de to spillerne, og husker at de skal spille `antall_spill` ganger mot hverandre.
- `arranger_enkeltspill`: For å arrangere et enkelt spill må systemet spørre begge spillere om hvilken aksjon de velger, sjekke hvem som vant, rapportere valgene og resultatet tilbake til spillerne, og gi en tekstuell beskrivelse av resultatet for eksempel i form av en enkelt linje:

```
Historiker(2): Stein. MestVanlig: Saks -> Historiker(2) vinner
```

Merk at mye av denne funksjonaliteten ligger i klassen `EnkeltSpill`, slik at `MangeSpill.arranger_enkeltspill` kan hente det meste av sin funksjonalitet derfra.

- `arranger_turnering`: Gjennomføre `antall_spill` enkelt-spill mellom de to spillerne. Rapportere gevinst-prosenten for hver av dem når turneringen er ferdig. Det

er også interessant (men optional) å vise hvordan gevinst-prosenten utvikler seg over tid (se Figure 2, som viser score for `Historiker(2)` mot `MestVanlig` over 100 spill). Det er enkelt å lage slike figurer ved å inkludere `pyplot` fra `matplotlib`.

2.3.2 GUI

Du skal også lage et grafisk grensesnitt der en `Spiller` spiller mot et menneske. Mennesket viser sine valg ved å trykke på knapper i grensesnittet. Vi bruker `Tkinter` for å generere det grafiske brukergrensesnittet.

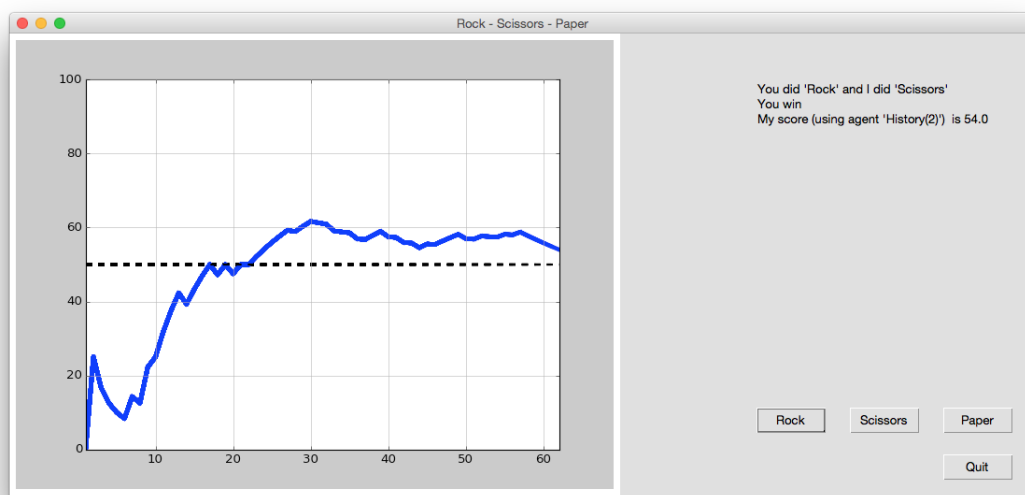


Figure 3: Eksempel på et GUI. Vi har knapper for å interagere med systemet, resultatet fra siste spill oppe til høyre, og en kurve som viser resultatene over tid på venstre side.

For å få dette til å virke må dere importere følgende:

```
from tkinter import Tk, BOTH, StringVar
from tkinter.ttk import Frame, Button, Label, Style
import pylab
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
```

I `Tkinter` defineres hvert visuelle objekt spesielt. Det å gjennomføre dette er litt tidkrevende, så vi oppgir et eksempel på kode som gir et grensesnitt som i Figure 3:

```

class GUITournament(Frame):
    # Klassen GUITournament definerer en turnering mellom menneske
    # og en Spiller
    spiller = None
    # Resultater holder resultatet av kampene - for plotting
    resultater = None
    # Denne labelen vil brukes for aa rapportere enkeltspill
    resultat_label = None

    def __init__(self, parent, motspiller):
        Frame.__init__(self, parent)
        self.parent = parent
        # Huske hvem vi spiller mot
        self.spiller = motspiller
        # Initiere listen av resultater
        self.resultater = []
        # Foreløpig ikke noe aa rapportere
        self.resultat_label = StringVar()
        self.resultat_label.set("Beskrivelse av siste spill kommer her")
        self.style = Style()
        self.fig = None

    def setup_gui(self):
        self.parent.title("Stein - Saks - Papir")
        self.style.theme_use("default")
        self.pack(fill=BOTH, expand=1)
        # Label for rapportering
        label = Label(self.parent, textvariable=self.resultat_label)
        label.place(x=800, y=50)
        # Buttons
        # Disse fyrer av metoden self.arranger_enkeltspill som er
        # definert i klassen. Denne metoden tar aksjonen til mennesket
        # som startup, og gjennomfører spillet
        # Samme type oppførsel for de tre aksjons-knappene
        rock_button = Button(self, text="Stein",
            command=lambda: self.arranger_enkeltspill(Aksjon("Stein")))
        rock_button.place(x=800, y=400)
        scissors_button = Button(self, text="Saks",
            command=lambda: self.arranger_enkeltspill(Aksjon("Saks")))
        scissors_button.place(x=900, y=400)
        paper_button = Button(self, text="Papir",
            command=lambda: self.arranger_enkeltspill(Aksjon("Papir")))
        paper_button.place(x=1000, y=400)

```

```

# quit_button avslutter GUI'et naar den trykkes
quit_button = Button(self, text="Quit", command=self.quit)
quit_button.place(x=1000, y=450)
# Embedde en graf i vinduet for aa rapportere fortloepende score
self.fig = FigureCanvasTkAgg(pylab.figure(), master=self)
self.fig.get_tk_widget().grid(column=0, row=0)
self.fig.show()

```

Som dere ser trenger vi metoden `arranger_enkeltspill` definert i denne klassen for at dette skal virke. Denne metoden får aksjonen valgt av bruker som input, spør den implementerte spilleren om dennes aksjon, finner ut hvem som vinner og rapporterer tilbake både til den implementerte spilleren og til bruker gjennom grensesnittet. For å rapportere tilbake resultater til grensesnittet kan vi bruke elementene som er definert der. For eksempel vil `self.resultat_label.set(txt)` sette innholdet av variabelen `txt` i tekst-området `resultat_label`, og denne kode-snutten

```

# Update plot
plt.figure(self.fig.figure.number) # Handle til figuren
plt.ion()
plt.plot(range(1, len(self.resultat) + 1),
         100 * cumsum(self.resultat) /
         range(1, len(self.resultat) + 1), 'b-', lw=4)
plt.ylim([0, 100])
plt.xlim([1, max(1.1, len(self.resultat))])
plt.plot(plt.xlim(), [50, 50], 'k--', lw=2)
plt.grid(b=True, which='both', color='0.65', linestyle='--')
self.fig.show()

```

plotter gjennomsnittlig suksess-rate dersom `self.resultat` brukes til å holde resultatet av hvert enkelt-spill. For at dette skal kjøre forutsettes at du har importert de nødvendige pakkene i forkant. Du trenger både `import matplotlib.pyplot as plt` og `from numpy import cumsum`.

Kode for å create vinduet og definere at kampen skal være mot `Historiker(2)`:

```

# Styrer spiller gjennom Tkinter/GUI.
root = Tk()
# Definer et vindu med gitte dimensjoner
root.geometry("1100x500+300+300")
# Lag instans, og kjoer oppsett av GUI (knapper etc)
GUITournament(root, Historiker(2)).setup_gui()
# Vis vindu, og utfoer tilhoerende kommandoer
root.mainloop()

```


3 Hva kreves for å få full score på oppgaven

For å bestå denne oppgaven må du:

- Løse den alene innen fristen – som er på 2 uker.
- Kode de 4 Spiller-klassene Sekvensiell, Tilfeldig, MestVanlig og Historiker.
- Implementere både grafisk og tekst-basert grensesnitt og bruke dem til å gjennomføre turneringer.

Ekstra utfordring – **ikke** obligatorisk:

Implementer din egen ide for en god Spiller. For å være “god” bør spilleren din oppnå minst 60% score mot Histroiker(2) over 1000 runder og minst minst 95% score mot Sekvensiell.