IN1000 Obligatorisk innlevering 7

Frist for innlevering: 23.10. kl 12:00

Introduksjon

I denne innleveringen skal du lage et program som simulerer cellers liv og død. Dette skal du gjøre ved hjelp av en modell kalt Conway's Game of Life (les mer om Game of Life her).

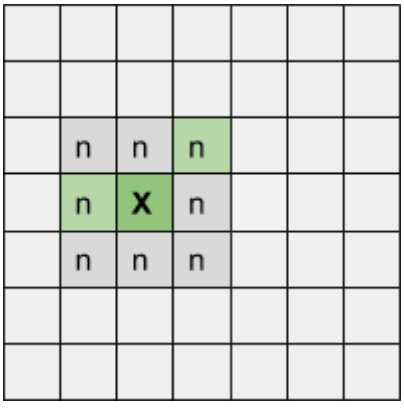
Kort fortalt skal programmet holde styr på et kvadratisk spillebrett av en vilkårlig størrelse, der hvert felt i brettet skal kunne inneholde en celle. En celle kan være levende eller død. Simuleringen utspiller seg gjennom flere generasjoner, der celler dør eller lever i en generasjon avhengig av sine omgivelser i den forrige.

Programmet skal la brukeren observere simuleringen generasjon for generasjon ved å tegne opp spillebrettet i terminalen sammen med tilleggsinformasjon om hvilken generasjon vi ser på samt hvor mange celler som for øyeblikket lever.

Når du leverer denne oppgaven skal du levere **alle** python-filer du har brukt i løsningen. Det er viktig at du kommenterer hvordan løsningen din fungerer. I tillegg **skal** du også levere en fil README.txt som beskriver hva som har vært utfordrende og hva som har vært enkelt. Dersom deler av programmet ikke fungerer skal du også beskrive *hva* som ikke fungerer, et forslag til *hvorfor* det ikke fungerer, samt en beskrivelse av hvordan du ville ha angrepet oppgaven annerledes dersom du fikk et nytt forsøk.

Spillets regler

En ny generasjon skapes ved at alle cellene i brettet endrer status avhengig av sine naboceller. Som naboceller regnes alle celler med felles hjørne eller kant, både levende og døde.



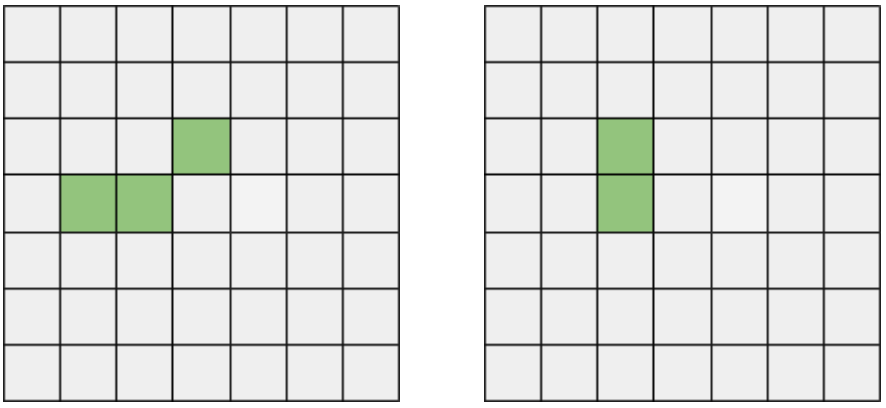
Illustrasjon 1: En celle (X) og dens naboceller (grønne celler er "levende")

En celles nye status bestemmes av følgende regler:

- Dersom cellens nåværende status er "levende":
  - Ved færre enn to levende naboceller dør cellen (*underpopulasjon*).
  - Ved to eller tre levende naboceller vil cellen leve videre.
  - Hvis cellen har mer enn tre levende naboceller vil den dø (*overpopulasjon*)

- Dersom cellen er “død”:
    - Cellens status blir “levende” (*reproduksjon*) dersom den har nøyaktig tre levende naboer.

NB! Hver celles tilstand er bare avhengig av naboene i forrige generasjon. Det betyr at neste tilstand til alle celler må bestemmes **før** man oppdaterer cellene.



Illustrasjon 2: Et eksempel på to etterfølgende generasjoner av celler i et 7x7 spillebrett

Struktur

Programmet består av tre filer med to klasser i tillegg til et “hovedprogram”, som alle skal leveres i hver sin kodefil. De beskrevne klassene skal navngis som beskrevet. Kodeskjelett for de tre filene er vedlagt oppgaven, i tillegg til noen ferdig implementerte metoder. Disse er det også mulig å implementere selv, for de som vil ha flere utfordringer. Det kan være nødvendig å utvide klassene med flere metoder.

Celle

Filnavn: *celle.py*

Klassen beskriver en celle i simuleringen. En celle skal ha en variabel som beskriver status (levende/død).

1. Skriv en konstruktør for klassen som oppretter cellen med status “død” som utgangspunkt.

2. Skriv metodene *settDoed* og *settLevende* som ikke tar noen parametere, men som setter statusen til cellen til henholdsvis “død” og “levende”. Oppdater konstruktøren slik at den tar i bruk metoden *settDoed*.

3. Skriv metoden *erLevende* som returnerer cellens status; *True* hvis cellen er levende og *False* ellers. Skriv i tillegg en metode som returnerer en tegnrepresentasjon av cellens status til bruk i tegning av brettet. Dersom cellen er “levende” skal det returneres en “O”, mens hvis den er død returneres et punktum.

Spillebrett

Filnavn: *spillebrett.py*

Denne klassen beskriver et todimensjonalt brett som inneholder celler. Spillebrettet skal holde styr på hvilke celler som skal endre status og oppdatere disse for hver generasjon.

1. Skriv ferdig konstruktøren for klassen Spillebrett. Konstruktøren tar imot dimensjoner på spillebrettet og lagrer disse i instansvariablene *self._rader* og *self._kolonner*. Konstruktøren skal:
    a. Opprette et *rutenett* i form av en todimensjonal (nøstet) liste. Rutenettet skal fylles med et antall Celle-objekter likt antall rader ganger antall kolonner.
    b. Opprette en variabel som holder styr på *generasjonsnummer* og som skal økes hver gang brettet oppdateres.

2. Utvid klassens konstruktør til å kalle på metoden *generer.* Denne metoden går gjennom rutenettet og sørger for at et tilfeldig antall celler får status "levende". Dette kalles et "seed" og utgjør utgangspunktet, eller "nulte generasjon" for cellesimuleringen vår. Denne metoden er ferdig implementert i kodeskjelettet vi har delt ut. En forklaring på implementasjonen kan du finne nederst i oppgaveteksten.

FRIVILLIG: Du kan implementere denne metoden selv. For å gjøre dette enklest mulig kan du ta i bruk Python-modulen *random*. Den inneholder blant annet funksjonen *randint(t1, t2)*, som tar imot to heltall og returnerer et tilfeldig tall mellom disse.

3. For å vise frem og teste spillebrettet skal du skrive metoden *tegnBrett*. Denne metoden skal bruke en nøstet for-løkke for å skrive ut hvert element i rutenettet. Husk å teste programmet ditt så langt ved å opprette et Spillbrett-objekt og skrive ut brettet i et lite testprogram. Tips til formatering av utskrift:
    a. For å unngå linjeskift etter hver utskrift kan du avslutte utskriften med en tom streng isteden, slik:

    ```
    print(arg, end="")
    ```

    b. Det kan være lurt å "tømme" terminalvinduet mellom hver utskrift. Dette kan du for eksempel gjøre ved å skrive ut et titalls blanke linjer før du skriver ut brettet.

4. For å bestemme hvilke celler som skal være "levende" og "døde" i neste generasjon trenger vi å vite statusen til hver celles nabo. Spillebrett-klassen inneholder derfor en metode *finnNabo*. Metoden tar imot to koordinater i rutenettet og returnerer en liste med alle cellens naboer. Denne metoden er ferdig implementert i kodeskjelettet vi har delt ut. En forklaring på implementasjonen kan du finne nederst i oppgaveteksten.

FRIVILLIG: Denne metoden kan du også implementere selv. Merk at metoden må ta høyde for at en celle ligger på en "kant" av brettet og kun legge til celler på plasser som ligger innenfor den todimensjonale listen.

5. For å beregne neste generasjon av celler trengs metoden *oppdatering*. Skriv denne metoden. Metoden skal gjøre følgende:
    a. Opprett to lister. Den ene listen skal inneholde alle døde celler som skal få status "levende", mens den andre skal inneholde levende celler som skal få status "død". La listene være tomme foreløpig.

    b. Deretter skal metoden gå gjennom rutenettet ved hjelp av en nøstet løkke. For hver celle skal den sjekke om cellen er levende eller død og deretter beregne om den skal endre status på bakgrunn av antallet levende naboer. Her blir du nødt til å hente opp alle naboene til en celle og telle antallet som lever. Følg listen med regler beskrevet under "Spillets regler" lenger opp. Celler som skal endre status skal legges inn i den riktige av de to listene vi laget tidligere.

    c. Først når alle cellene er sjekket og listene er fylt med Celle-objekter skal selve oppdateringen skje. Endre status på objektene i de to listene ved hjelp av Celle-metodene *settLevende* eller *settDoed.*

    d. Til sist må du huske å oppdatere telleren for antall generasjoner.

    e. Utvid testprogrammet fra deloppgave 3 ved å kalle på metoden *oppdatering* en gang. Oppfører programmet seg som forventet? Tips: Denne metoden kan testes ved å fylle rutenettet med et kjent mønster og se at det endrer seg som forventet etter en generasjon.

6. I tillegg til generasjonsnummer er vi interessert i den generelle cellestatusen på spillebrettet. Du skal derfor skrive en metode *finnAntallLevende* som kan beregne og returnere antallet levende celler. Dette kan du enklest gjøre ved å gå gjennom rutenettet og øke en teller for hver levende celle du finner.

Hovedprogram

Filnavn: *main.py*

Skriv et hovedprogram, *main*, der brukeren først skal bli spurt om å oppgi dimensjoner på spillebrettet. Deretter skal du opprette brettet og skrive ut den "nulte" generasjonen.

Illustrasjon 4: Eksempel på utskrift av spillebrettet

Ved hjelp en menyløkke og input skal brukeren deretter kunne velge å oppgi en tom linje for å gå videre til neste steg, eller skrive inn bokstaven “q” for å avslutte programmet. Hver gang brukeren oppgir at de ønsker å fortsette skal du kalle på *oppdatering*-metoden og deretter skrive ut brettet på nytt sammen med en linje som beskriver hvilken generasjon som vises og hvor mange celler som lever for øyeblikket.

Vedlegg 1: Kode

Celle

Filnavn: *celle.py*

```
class Celle:
    # Constructor
    def __init__(self):
        pass

    # Endre status
    def settDoed(self):
        pass

    def settLevende(self):
        pass
```

```
    # Hente status
    def erLevende(self):
        pass

    def hentStatusTegn(self):
        pass
```

Spillebrett

Filnavn: *spillebrett.py*

```
from random import randint
from celle import Celle

class Spillebrett:
    def __init__(self, rader, kolonner):
        self._rader = rader
        self._kolonner = kolonner
        self._rutenett = []
        # ...

    def tegnBrett(self):
         pass

    def oppdatering(self):
         pass

    def finnAntallLevende(self):
         pass

    def generer(self) :
        for i in range (self._rader):
            for j in range (self._kolonner):
                rand = randint(0,3)
                if rand == 3 :
                    self._rutenett[i][j].settLevende()

    def finnNabo(self, x, y):
        naboliste = []
        for i in range (-1, 2):
            for j in range (-1, 2):
                naboX = x+i
                naboY = y+j
                if (naboX == x and naboY == y) != True:
                    if (naboX < 0 or naboY < 0 or naboX > self._kolonner-1 or
    naboY > self._rader-1) != True:
                        naboliste.append(self._rutenett[naboY][naboX])
    return naboliste
```

Hovedprogram

Filnavn: *main.py*

```
from spillebrett import Spillebrett

def main():
    pass

# starte hovedprogrammet
main()
```

Vedlegg 2: Forklaring på utgitt kode

a) Hvordan generere "generasjon 0":

Øverst i dokumentet må du inkludere følgende linje:

```
from random import randint
```

Selve metoden kan defineres slik:

```
def generer(self) :
    for i in range (self._rader):
        for j in range (self._kolonner):
            rand = randint(0,3)
            if rand == 3 :
                self._rutenett[i][j].settLevende()
```

Denne metoden genererer et tilfeldig tall for hver celle (her mellom 0 og 3) og sammenlikner det med et statisk sjekktall (her 3). Dersom tallene matcher vil cellen settes til "levende". Test gjerne ut andre verdier og se hvordan det påvirker antallet levende celler i første generasjon. Er du ekstra interessert kan du se eksempler på spesifikke mønstre her.

b) Finn nabo i Spillebrett

```
def finnNabo(self, x, y):
    naboliste = []
    for i in range (-1, 2):
        for j in range (-1, 2):
            naboX = x+i
            naboY = y+j
            if (naboX == x and naboY == y) != True:
                if (naboX < 0 or naboY < 0 or naboX > self._kolonner-1 or
naboY > self._rader-1) != True:
                    naboliste.append(self._rutenett[naboY][naboX])
    return naboliste
```

Metoden går gjennom de ni mulige plassene rundt en celle, men luker ut senteret (som er selve cellen og ikke en nabo), samt plasseringer som er for små eller for store til å eksistere i

spillebrettet (utenfor kanten av brettet). Alle gyldige naboer legges i en liste av naboer som returneres.