

IN1000 Obligatorisk innlevering 8 - 2020

Frist for innlevering: 03.11. kl 23:59

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 spillebrett av en vilkårlig størrelse, der hvert felt i brettet inneholder en celle. En celle kan være levende eller død. Simuleringen utspiller seg gjennom flere generasjoner ved hjelp av jevnlig oppdateringer, der celler dør eller lever avhengig sine omgivelser.

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.

Noen tips til klassen Spillebrett vil gjennomgås i plenumsvideoen og gruppetimene i uke 10 (kalenderuke 43/44). Før det bør du ha satt deg inn i spillet og skrevet klassen Celle.

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 kommentere i Devilry på 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 berørende celler, både levende og døde.

		n	n	n			
		n	X	n			
		n	n	n			

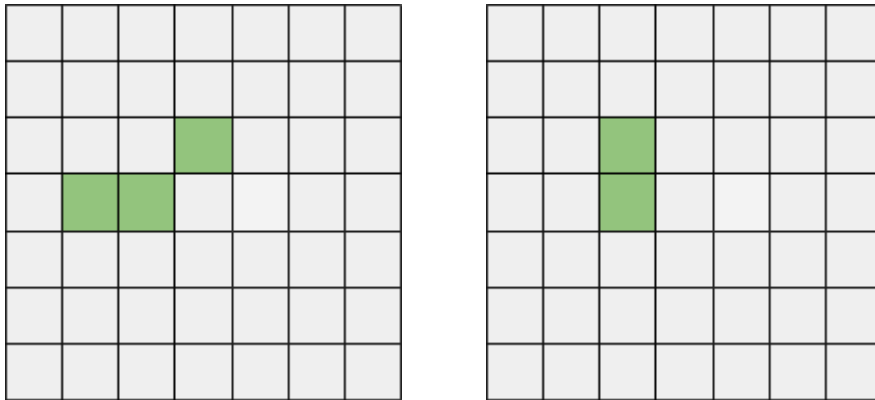
Illustrasjon 1: En celle (X) og dens naboceller (grønne celler er "levende"). X har altså 8 naboceller, og 2 av dem er levende.

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.

Merk at oppdateringen av cellenes status skjer **samtidig!** Det betyr at vi må bestemme ny status på alle celler avhengig av nåværende status *før* oppdateringen faktisk skjer.

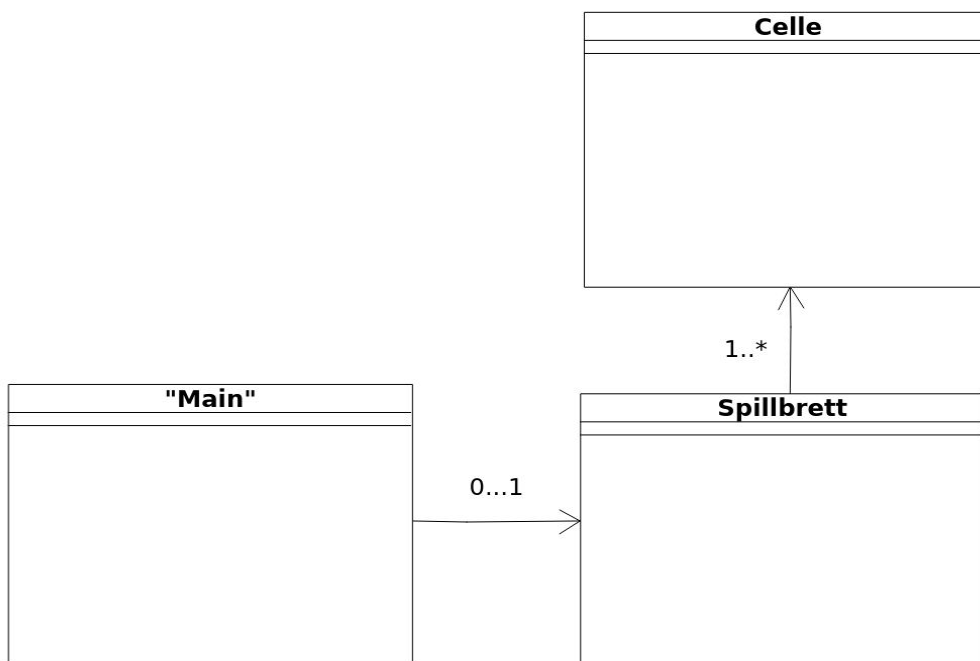


Illustrasjon 2: To generasjoner av celler i et 7*7-størrelse spillebrett

Struktur

Programmet består to klasser og et “hovedprogram”, som alle skal leveres i hver sin kodefil. Kodeskjelett for de tre filene er vedlagt oppgaven. Det kan være nødvendig å utvide kodeskjelettene med flere metoder.

Nedenfor kan du se et UML klassediagram av datastrukturen. Klassediagrammet viser at klassen Spillbrett refererer til 1 eller flere instanser (objekter) av klassen Celle, mens hovedprogrammet skal referere til null eller ett instans av klassen Spillbrett.



Celle

Filnavn: *celle.py*

Klassen beskriver en celle i simuleringen. En celle skal ha en variabel som beskriver status (levende/død). Se vedlegg i slutten av dokumentet for kodeskjelett.

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”.
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. Metoden *_generer* 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.
 - a. Importer *random* i programmet ditt. Opprett metoden *_generer*, som skal gjøre slik at hver celle i rutenettet har $\frac{1}{3}$ sjanse for å være levende. *Random* har en metode *randint(tall1, tall2)* som returnerer et tilfeldig tall mellom disse. Den regner fra og med *tall1*, til og med *tall2*. Eksempel:

```
random.randint(0,2)
```

Dette kallet returnerer et tilfeldig tall fra og med 0, til og med 2, altså 0, 1 eller 2.
 - b. Utvid klassens konstruktør til å kalle på metoden *_generer*.
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 Spillebrett-objekt og skrive ut

brettet i et lite testprogram. Dette kan du gjøre i metoden *main* i fila *main.py* beskrevet lenger ned i oppgaveteksten. 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 skal ta imot en celles koordinater (rad og kolonne) i rutenettet og returnere en liste med alle cellens naboer. Se illustrasjon 1 og 3 som viser to ulike celler med naboer. Pass på kanter og hjørner.

X	n					
n	n					

Illustrasjon 3: Cellen X har tre naboer, to døde og én levende.

5. For å beregne neste generasjon av celler trengs metoden *oppdatering*. Denne 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 reglene beskrevet under “Spilletts 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 skjer selve oppdateringen, og objekter i de to listene endrer status 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

Vedlegg 1: Kodeskjeletter

Her brukes ordet *pass* i koden bare som plassholder så du kan teste programmet ditt før du har skrevet alle metodene. Det skal ikke være med når du leverer. PS: Om du kopierer koden direkte fra PDF'en kan du få formateringsfeil.

Celle

Filnavn: celle.py

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

    # Endre status
    def settDoed(self):
        pass

    def settLevende(self):
        pass

    # Hente status
    def erLevende(self):
        pass

    def hentStatusTegn(self):
        pass
```

(Kodeskjeletter fortsetter på neste side)

Spillebrett

Filnavn : spillebrett.py

```
from random import randint
from celle import Celle

class Spillebrett:
    def __init__(self, rader, kolonner):
        pass

    def tegnBrett(self):
        pass

    def oppdatering(self):
        pass

    def finnAntallLevende(self):
        pass

    def _generer(self) :
        pass

    def finnNabo(self, rad, kolonne):
        pass
```

Main

Filnavn: main.py

```
from spillebrett import Spillebrett

def main():
    pass

# starte hovedprogrammet
main()
```