IN1000 Game of Life - 2022

Introduksjon

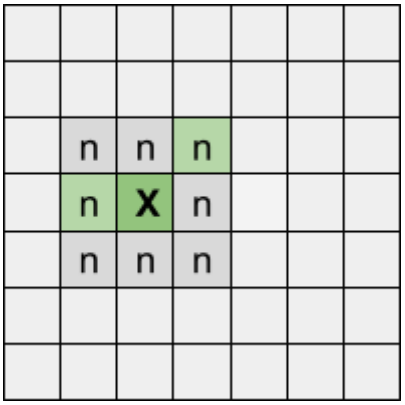
I denne innleveringen skal du lage et program som simulerer cellers liv og død. Dette skal du gjøre ved hjelp av en modell kalt Conway's Game of Life (les mer om Game of Life her).

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 jevnlige 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.

Spillets regler

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

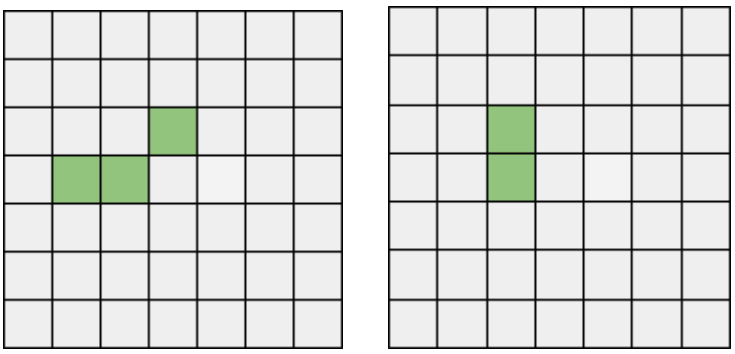


Illustrasjon 1: En celle (X) og dens naboceller (grønne celler er "levende"). X har altså 8 naboceller, og 2 av dem er levende.

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.



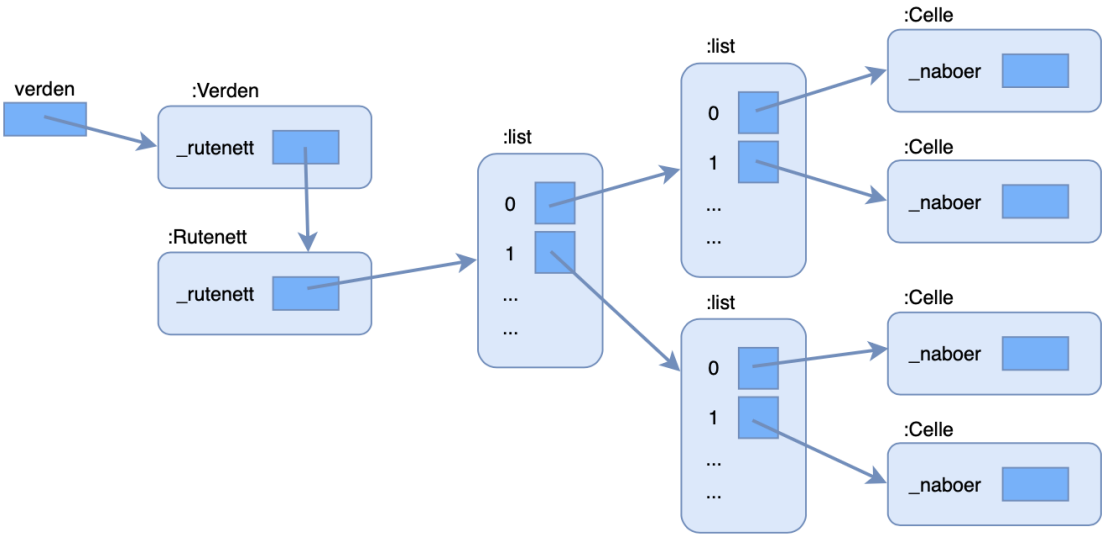
*Illustrasjon 2: To generasjoner av celler i et 7*7-størrelse spillebrett*

Struktur

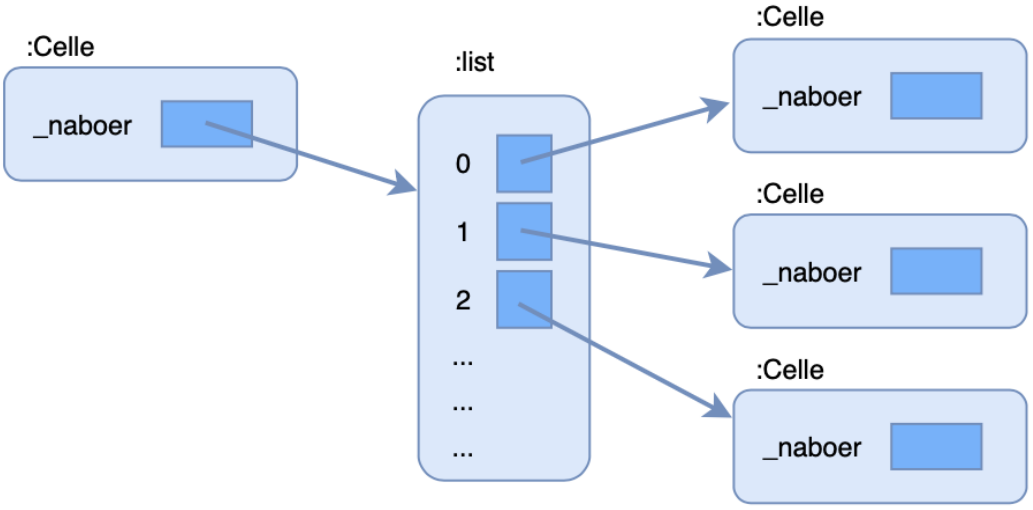
Programmet består tre klasser og et "hovedprogram", som alle skal leveres i hver sin kodefil. **Kodeskjelett** for de tre filene er vedlagt oppgaven. Her brukes ordet *pass* i koden bare som plassholder så du kan teste programmet ditt før du har skrevet alle metodene. Det skal ikke være med når du leverer. Det kan være nødvendig å utvide kodeskjelettene med flere metoder.

Det følger også med **test-filer** for klassene *Celle*, *Rutenett*, og *Verden*. Her kan man kalle på ulike test-funksjoner som skriver "Alt riktig" eller en beskjed om hvilken av metodene over som ikke gjør det den skal. Når man har gjort en deloppgave så trenger man kun å fjerne kommentartegnet for den aktuelle deloppgave-testen, og se at alt virker. Man trenger ikke å levere test-filene.

Nedenfor kan du se hvordan datastrukturen kan se ut under kjøring. Hovedprogrammet skal referere en instans av klassen *Verden,* som igjen refererer til en instans av klassen Rutenett. I denne instansen er det en nøstet liste med referanser til instanser av klassen Celle, hvor hver celle kommer til å ha en liste med referanser til andre instanser av klassen Celle (naboer).



*Datastrukturen for hvordan **hovedprogram** vil se ut under kjøring*



*Datastrukturen for hvordan en instans av klassen **Celle** vil se ut under kjøring*

Celle

Filnavn: *celle.py*

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

Det er lagt ved en fil "*test_celle.py*" som inneholder ferdige funksjoner som kan brukes til å teste metodene man skriver underveis.

1. Skriv en **konstruktør** for klassen som oppretter cellen med _status "doed" som utgangspunkt. Skriv også instansvariablene *_naboer* som settes lik en tom liste, og *_ant_levende_naboer* som settes lik 0

   Test-funksjon: ***test_celle***

2. Skriv metodene ***sett_doed*** og ***sett_levende*** som ikke tar noen parametere, men som setter statusen til cellen til henholdsvis "doed" og "levende".

   Test-funksjon: **test_*sett_doed_levende***

3. Skriv metoden ***er_levende*** som returnerer cellens status; *True* hvis cellen er levende og *False* ellers.

   Test-funksjon: **test_*er_levende***

4. Skriv metoden ***hent_status_tegn*** 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.

Test-funksjon: **test_hent_status_tegn**

5. Skriv metoden ***legg_til_nabo*** som har en instans av klassen Celle som parameter (nabo), og legger nabo til i listen *_naboer*

   Test-funksjon: **test_legg_til_nabo**

6. I tillegg trenger vi følgende metoder når vi skal oppdatere statusen for en celle:
   - **tell_levende_naboer** som går gjennom *_naboer* og teller antall levende naboer. Husk å oppdatere instansvariabelen *_ant_levende_naboer*
   - **oppdater_status** som ut fra Spillets regler (se avsnittet lenger opp) endrer statusen til en celle basert på antall levende naboer

   Test-funksjon: **test_tell_levende_naboer**
   Test-funksjon: **test_oppdater_status**

Rutenett

Filnavn: *rutenett.py*

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

Filen "*test_rutenett.py*" inneholder ferdige funksjoner som kan brukes til å teste metodene.

1. Skriv **konstruktøren** for klassen Rutenett. Konstruktøren tar imot dimensjoner på rutenettet og lagrer disse i instansvariablene *self._ant_rader* og *self._ant_kolonner*.

   Test-funksjon: **test_konstruktoer_uten_rutenett**

2. I konstruktøren ønsker vi også å lage en ny instansvariabel *_rutenett* i form av en todimensjonal (nøstet) liste. Rutenettet skal fylles med et antall instanser av klassen *Celle* likt antall rader ganger antall kolonner. For å lage dette rutenettet trenger vi følgende metoder:
   a. Metoden **_lag_tom_rad** som bare lager en enkel liste med like mange None-verdier som det skal være kolonner, og returnerer denne listen.
   b. Metoden **_lag_tomt_rutenett** som setter rader laget av *_lag_tom_rad* sammen i en ytre liste.

   Deretter skal man i konstruktøren kalle de riktige metodene inne i konstruktøren for å opprette rutenettet (uten celler).

   Test-funksjon: **test_konstruktoer_med_rutenett**

3. Metoden **fyll_med_tilfeldige_celler** 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. Her skal man bruke en nøstet for-løkke, og så for hver "plass" i rutenettet skal man:
    b. Kalle på metoden **lag_celle**, som oppretter en instans av klassen Celle og legger det inn på en plass i den nøstede listen ut fra *rad* og *kol.* Hver celle i rutenettet har ⅓ sjanse for å være levende når de skal legge inn i rutenettet. Her kan man bruke metoden *sett_levende* fra klassen Celle.
    c. For å tilfeldig bestemme om en celle skal være levende kan man importere *random* i programmet sitt, ***from random import randint***. Random har en metode randint(tall1, tall2) som returnerer et tilfeldig tall mellom disse. Eksempel: *randint(0,2)*. Dette kallet returnerer et tilfeldig tall fra og med 0, til og med 2, altså 0, 1 eller 2.

    Test-funksjon: **test_fyll_med_tilfeldige_celler**

4. Skriv metoden ***hent_celle*** som tar imot en celles koordinater (rad og kolonne) i rutenettet, og returnerer cellen til den gitte posisjonen. Hvis en ulovlig rad- eller kolonneindeks er gitt (for lav eller for høy indeks), så skal metoden returnere None.

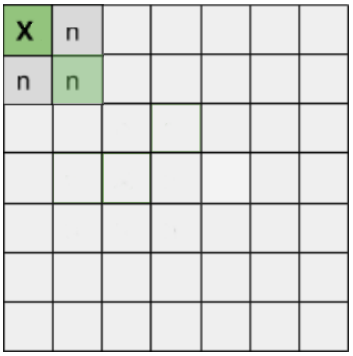
    Test-funksjon: **test_hent_celle**

5. For å vise frem og teste rutenettet skal du skrive metoden ***tegn_rutenett***. Denne metoden skal bruke en nøstet for-løkke for å skrive ut hvert element i rutenettet. 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.

    Test-funksjon: **test_tegn_rutenett**

6. For å bestemme hvilke celler som skal være "levende" og "døde" i neste generasjon trenger vi å vite statusen til hver celles nabo. Rutenett-klassen inneholder derfor en metode ***_sett_naboer***. Metoden skal ta i mot en celles koordinater og sette referanser til alle instanser av klassen *Celle* som er nabo for den gitte Celle instansen.
    - Husk at du kan bruke *hent_celle* til å hente celler uten å få feilmeldinger for ulovlige indekser. Det vil si, om du f.eks. kaller *hent_celle og sender -1 som argument for rad eller noe slikt, er det ikke værre enn at du får None tilbake (og kan enkelt sjekke for det)* Se illustrasjon 1 og 3 som viser to ulike celler med naboer.



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

- Pass på å sjekke at alle naboene til en Celle er riktige. Her kan man printe ut (rad,kol) for alle naboene til en gitt Celle og sjekke at indeksene er riktige. På illustasjonen under så har *Celle X (0,0)* naboer med plass *(0,1), (1,0)* og *(1,1)*. Pass på kanter og hjørner, og at man ikke legger til den gitte cellen som nabo til seg selv.

Test-funksjon: **test_sett_naboer**

7. Skriv en metode ***koble_celler*** som ved hjelp av en nøstet for-løkke skal kalle på *_sett_naboer* for hver plass (rad, kol) i rutenettet.

   Test-funksjon: **test_koble_celler**

8. I tillegg trenger vi to andre metoder som skal brukes senere i klassen *Verden*
   - **hent_alle_celler** skal returnere en enkel liste (ikke nøstet) med alle instanser av klassen *Celle* i et rutenett.
   - **antall_levende** skal returnere antall levende celler i rutenettet. Dette kan du enklest gjøre ved å gå gjennom rutenettet og øke en teller for hver levende celle du finner.

   Test-funksjon: **test_hent_alle_celler**
   Test-funksjon: **test_antall_levende**

Verden

Filnavn: *verden.py*

I denne klassen skal vi oppdatere rutenettet og skrive ut hver generasjon av simuleringen.

Filen "*test_verden.py*" inneholder ferdige funksjoner som kan brukes til å teste metodene.

1. Skriv konstruktøren for klassen *Verden*. Den skal ta inn antall rader og kolonner som parametere, og opprette en instans av klassen Rutenett som lagres i instansvariabelen *_rutenett*. Klassen trenger å holde styr på antall generasjoner og trenger derfor å ha en instansvariabel *_generasjonsnummer* som skal settes til lik 0. I tillegg så skal man fylle rutenettet med tilfeldige celler og koble cellene sammen.

Tips: bruk metoder fra *Rutenett* klassen.

Test-funksjon: **test_konstruktoer**

2. Skriv metoden **tegn** skal tegne rutenettet (printe ut) i tillegg til å skrive ut generasjonsnummeret og antall levende celler som er igjen.

   Test-funksjon: **test_tegn**

3. Skriv metoden **oppdatering**, som har tre ansvarsområder (hint: gjør bruk av eksisterende metoder):
   a. Gå gjennom alle celler i rutenettet og telle levende naboer for hver celle
   b. Gå gjennom alle celler i rutenettet på nytt, og oppdatere status på hver celle
   c. Til sist må du huske å oppdatere telleren for antall generasjoner.

   Test-funksjon: **test_oppdatering**

4. Utvid *hovedprogram* i *hovedprogram.py* til å kalle på *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. Eksempler på mønstre av ulike typer finner du under overskriften "Examples of patterns" her. Du kan for eksempel velge et mønster som gjentar seg i et visst antall generasjoner - slike kalles *oscillators*.

Hovedprogram

Filnavn: *hovedprogram.py*

Skriv et hovedprogram, der brukeren først skal bli spurt om å oppgi dimensjoner på spillebrettet. Deretter skal du opprette "verden" og tegne "nulte" generasjonen.

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 tegne verden på nytt. Her skal man også ha en linje som beskriver hvilken generasjon som vises og hvor mange celler som lever for øyeblikket.

Eksempel på utskrift av spillebrettet