Sist endret: 09.04.2025

IN1000 Innlevering 8: Game of Life

Innleveringsfrist: tirsdag 29. april 2025, kl. 23:59

Introduksjon

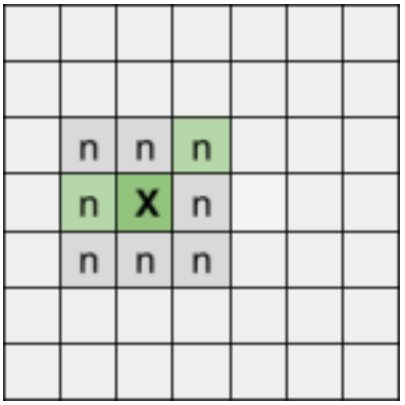
I denne innleveringen skal du lage et program som simulerer cellers liv og død. Dette skal du gjøre ved hjelp av en modell kalt Conway's Game of Life (les mer om Game of Life).

Kort fortalt skal programmet holde styr på et spillebrett av en vilkårlig (hvilken som helst) størrelse, der hvert felt i brettet inneholder en celle. En celle kan være levende eller død. Simuleringen utspiller seg over flere generasjoner. Hver generasjon fremkommer gjennom at spillebrettet oppdateres jevnlig. En oppdatering skjer ved at celler lever eller dør avhengig av sine omgivelser.

Programmet skal la brukeren observere simuleringen generasjon for generasjon. Derfor skal spillebrettet tegnes opp (skrives ut) i terminalen for hver generasjon, sammen med tilleggs informasjon om hvilken generasjon vi ser på. I tillegg skal det for hver generasjon skrives ut i terminalen antallet 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.



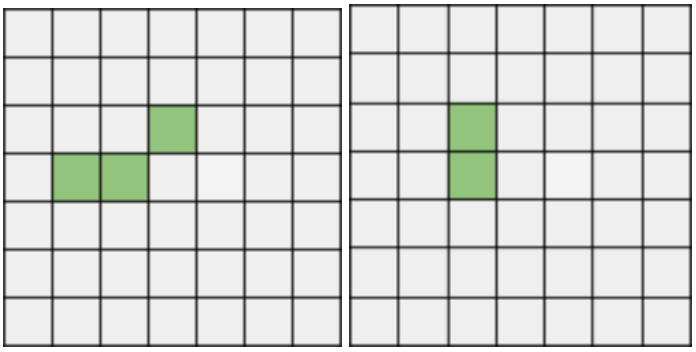
Illustrasjon 1: En celle (X) og dens naboceller (grønne celler er "levende"). X har altså 8 naboceller, og 2 av dem er levende.

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 generasjonsskiftet skjer samtidig *for alle celler*. Det betyr at du må ta vare på informasjon om status for hver celles naboceller *før* noen av cellene oppdateres. Oppdateringene fra en generasjon til en neste gjøres derfor i to faser i programmet: **1)** For hver celle beregner man hva man senere (i neste fase) skal sette dens status til (basert på den nåværende statusen til nabocellene), **2)** cellenes status oppdateres.



Illustrasjon 2: To generasjoner av celler i et spillebrett med 7 rader og kolonner.

Struktur og klassedesign

Programmet består av tre klasser og et **hovedprogram**, som alle skal leveres i hver sin kodefil. Klassen **Celle** holder på informasjon om en celles tilstand, naboene og antall naboer som lever. Den tilbyr metoder for å lese av og oppdatere informasjonen i ulike situasjoner.

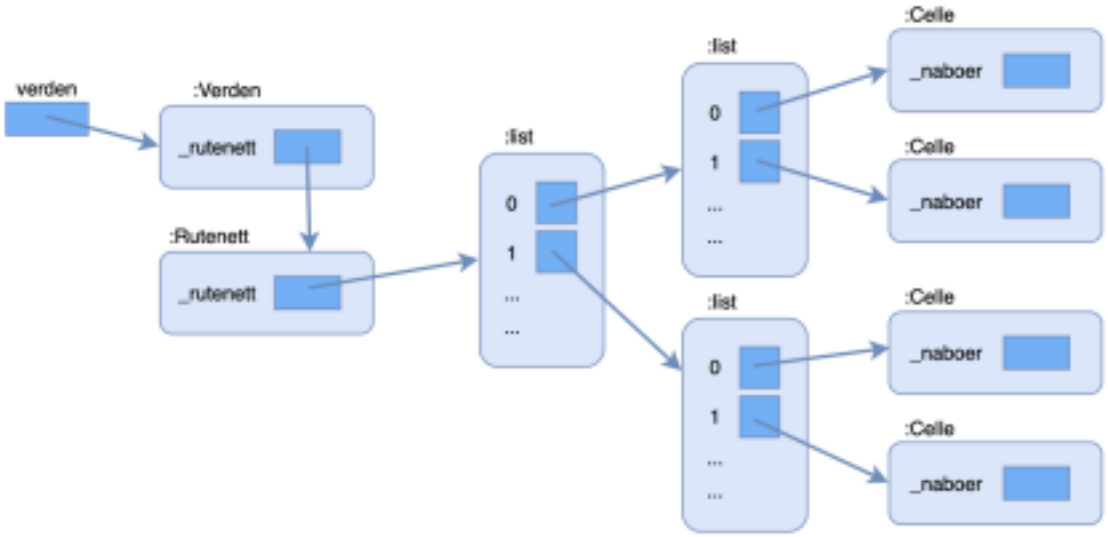
Klassen **Rutenett** holder på en nøstet liste (liste av lister) med celler. Den tilbyr metoder som bygger opp, oppdaterer og tegner rutenettet. Videre har klassen en metode for å hente ut en enkel (flat) liste med alle cellene.

Klassen **Verden** tilbyr metoder som oppretter, oppdaterer og tegner brettet ved å kalle på metodene i Rutenett og Celle.

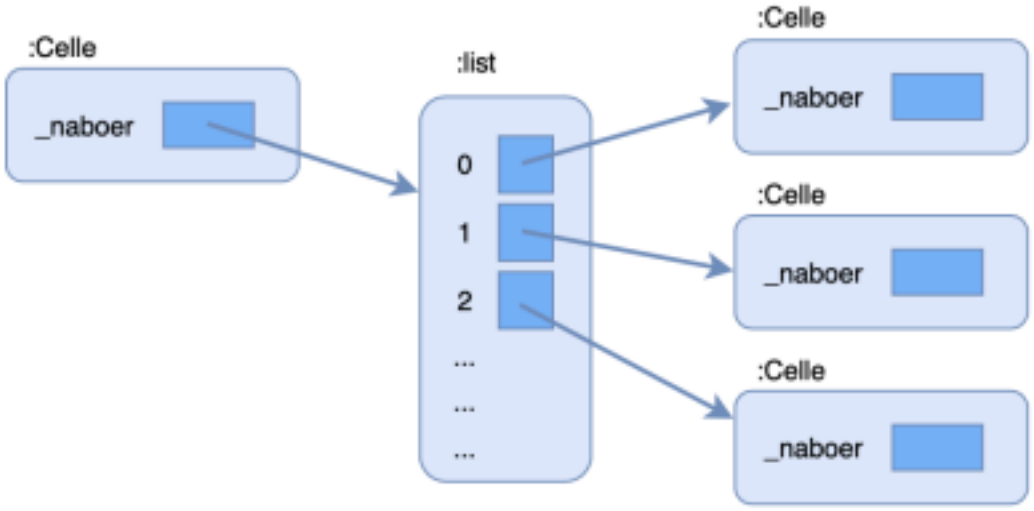
Kodeskjelettene for **Celle**, **Rutenett** og **Verden** er vedlagt oppgaven. Her brukes ordet *pass* i koden bare som plassholder, slik at du kan teste programmet ditt før du har skrevet alle metodene. Nøkkelordet pass skal ikke med når du leverer. Det kan være nødvendig å utvide kodeskjelettene med flere metoder.

Det følger også med *testfiler* for klassene **Celle**, **Rutenett** og **Verden**. I testfilene kan du kalle på ulike testfunksjoner som skriver ut *"Alt riktig"* i terminalen, eller skriver ut en beskjed hvis en metode ikke gjør det den skal. Når du har gjort en deloppgave, trenger du kun å fjerne kommentartegnet for den aktuelle testen for en deloppgave. Sjekk at det ikke blir funnet feil i den. Merk: Testfilene skal leveres uendret sammen med de andre filene.

Nedenfor kan du se hvordan datastrukturen kan se ut under kjøring. Hovedprogrammet skal referere til en instans (et objekt) av klassen **Verden**, som igjen refererer til en instans av klassen **Rutenett**. I instansen av Rutenett er det en nøstet liste med celler, hvor hver celle har en liste med referanser til andre instanser av klassen **Celle**. Disse utgjør cellens 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 har blant annet en instansvariabel som beskriver dens status (levende/død).

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

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

Testfunksjon: **test_celle**

2. Skriv metodene **sett_doed** og **sett_levende.** Disse tar ingen parametere, men setter cellens status til henholdsvis “doed” eller “levende”.

   Testfunksjon: **test_sett_doed_levende**

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

   Testfunksjon: **test_er_levende**

4. Skriv metoden **hent_status_tegn,** som returnerer en tegnrepresentasjon av cellens status til bruk når brettet tegnes. Om cellen er “levende” skal metoden returnere “O”. Er cellen død, skal metoden returnere et punktum.

   Testfunksjon: **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* refererer til.

   Testfunksjon: **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 endrer statusen til en celle basert på antall levende naboer. Metoden må ta hensyn til spillets regler (se avsnittet lenger opp).

   Testfunksjoner: **test_tell_levende_naboer** og **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 status 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 i mot dimensjoner for rutenettet og lagrer disse i instansvariablene *_ant_rader* og *_ant_kolonner*.

   Testfunksjon: **test_konstruktoer_uten_rutenett**

2. I konstruktøren ønsker vi også å lage en ny instansvariabel *_rutenett.* Den skal referere til en todimensjonal (nøstet) liste. Rutenettet skal fylles med et antall instanser av klassen **Celle**, likt antall rader ganger antall kolonner. For å lage

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 du i konstruktøren kalle metodene som trengs for å opprette det tomme rutenettet (uten celler).

Testfunksjon: **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. Med en nøstet for-løkke, skal du for hver "plass" i rutenettet:

- 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 den legges inn i rutenettet. Her kan du bruke metoden **sett_levende** fra klassen **Celle**.
- For å tilfeldig bestemme om en celle skal være levende, kan man importere *random* i programmet sitt, med *from random import randint*. Random har en metode *randint(tall1, tall2)*, som returnerer et tilfeldig tall mellom tall1 og tall2. Eksempel: *randint(0,2)* returnerer et tilfeldig tall fra og med 0 til og med 2, altså 0, 1 eller 2.

Testfunksjon: **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), skal metoden returnere *None*.

Testfunksjon: **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:

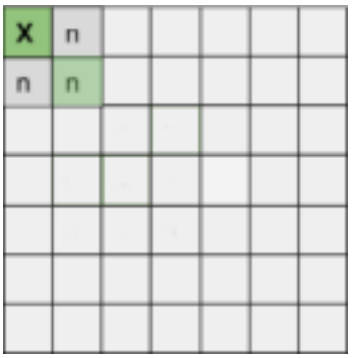
- For å unngå linjeskift etter hver utskrift kan du avslutte utskriften med en tom streng isteden, slik: *print(ditt_argument, end="")*.

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

Testfunksjon: **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 til den gitte cellen.

● 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 lignende, er det ikke verre enn at du får *None* tilbake (og kan enkelt sjekke for det). Illustrasjon 1 og 3 viser to celler med naboer.



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

● Sjekk at alle naboene til en celle er riktige. Her kan du printe ut (rad,kol) for alle naboene til en gitt celle og sjekke at indeksene er riktige. I illustrasjonen over har celle *X (0,0)* naboer med plass *(0,1), (1,0)* og *(1,1)*. Pass på kanter og hjørner, og at du ikke legger til den gitte cellen som nabo til seg selv.

Testfunksjon: **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.

Testfunksjon: **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 rutenettet.
● **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.

Testfunksjoner: **test_hent_alle_celler** og **test_antall_levende**

Verden

Filnavn: *verden.py*

I denne klassen skal du 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, 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 settes lik 0. I tillegg skal du fylle rutenettet med tilfeldige celler og koble cellene sammen. Tips: bruk metoder fra **Rutenett**-klassen.

Testfunksjon: **test_konstruktoer**

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

Testfunksjon: **test_tegn**

3. Skriv metoden **oppdatering**, som har tre ansvarsområder:

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 slutt må du huske å oppdatere generasjonsnummeret.

Hint: bruk eksisterende metoder i **Rutenett** og **Celle** når du løser a, b og

c. Testfunksjon: **test_oppdatering**

Hovedprogram

Filnavn: *hovedprogram.py*

Skriv et **hovedprogram**, der brukeren først skal bli spurt om å oppgi antall rader og kolonner spillebrettet skal ha. Deretter skal du opprette “verden” og tegne den “0-te” generasjonen.

Ved hjelp av en løkke 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 vedkommende ønsker å fortsette, skal du kalle på **oppdatering**-metoden og deretter tegne verden på nytt. Utskriften skal også inkludere nummeret til generasjonen som vises og antall celler som lever i den generasjonen.

Sjekk at cellene i spillebrettet endrer seg etter spillereglene for hver nye generasjon. Under ser du et eksempel på utskrift av et spillebrett.

Eksempel på utskrift av et spillebrett



Krav til innlevering

- Kun *.py*-filene, inkludert testfilene, skal leveres inn. Testfilene **skal** være uendrede.
- Spørsmålene til innleveringen skal besvares i kommentarfeltet.
- Koden skal inneholde gode kommentarer som forklarer hva programmet gjør og kommentarer som forklarer hvordan du har tenkt underveis.
- Programmet skal gi utskrifter som er enkle å forstå for brukere.

Hvordan levere oppgaven

1. Svar på følgende spørsmål i kommentarfeltet i Devilry. Spørsmålene **skal** besvares.
   - **a.** Hvordan synes du innleveringen var? Hva var enkelt og hva var vanskelig?
   - **b.** Hvor lang tid (ca) brukte du på innleveringen?
   - **c.** Var det noen oppgaver du ikke fikk til? Hvis ja:
     - **i.** Hvilke(n) oppgave er det som ikke fungerer i innleveringen?
     - **ii.** Hvorfor tror du at oppgaven ikke fungerer?
     - **iii.** Hva ville du gjort for å få oppgaven til å fungere hvis du hadde mer tid?
     - **iv.** Hva vil du ha tilbakemelding på?

2. Logg inn på Devilry.

3. Lever alle *.py*-filene, inkludert de uendrede testfilene, og svar på spørsmålene ovenfor i kommentarfeltet.

4. Husk å trykke på *Add delivery or question*. Sjekk deretter at innleveringen din er komplett. Du kan levere så mange ganger du vil frem til fristen. Det er ikke mulig å slette innleveringer i Devilry. **Alle** filer må være med hver gang du leverer, og filene må ha samme navn som oppgitt i oppgaveteksten. Retteren din vil gi tilbakemelding på kun den siste innleveringen, og derfor er det viktig at alle filer er med i siste innlevering.

5. Innleveringen gir deg mulighet for en konstruktiv tilbakemelding på hva du har fått til, i tillegg til forbedringer/misforståelser og om du er omtrent i rute med faget. De fleste vil ha behov for å løse mange flere oppgaver enn de du finner i denne innleveringen. Det får du anledning til i gruppetimene og i Trix. Dessuten inneholder læreboken mange
oppgaver (noen går litt utenfor vårt pensum). Du finner dessuten mye på nett - og ikke minst kan du finne på egne programmer og utvidelser til oppgaver.