IN1000 Game of Life - 2021

Introduksjon

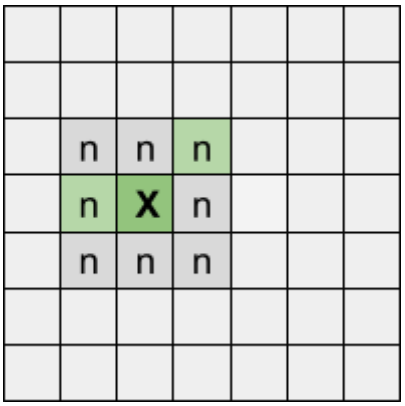
I denne innleveringen skal du lage et program som simulerer cellers liv og død. Dette skal du gjøre ved hjelp av en modell kalt Conway's Game of Life (les mer om Game of Life her).

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 berørende celler, både levende og døde.

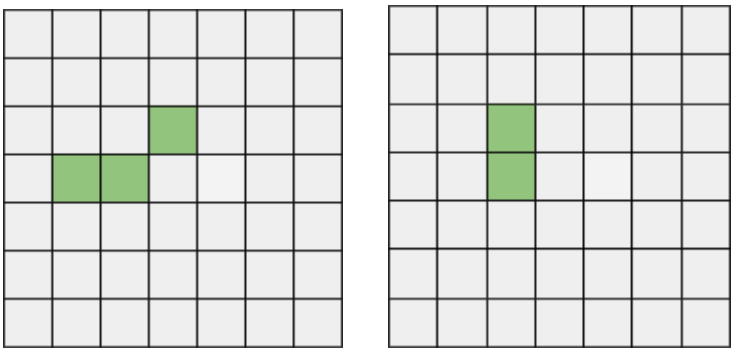


Illustrasjon 1: En celle (X) og dens naboceller (grønne celler er "levende"). X har altså 8 naboceller, og 2 av dem er levende.

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.

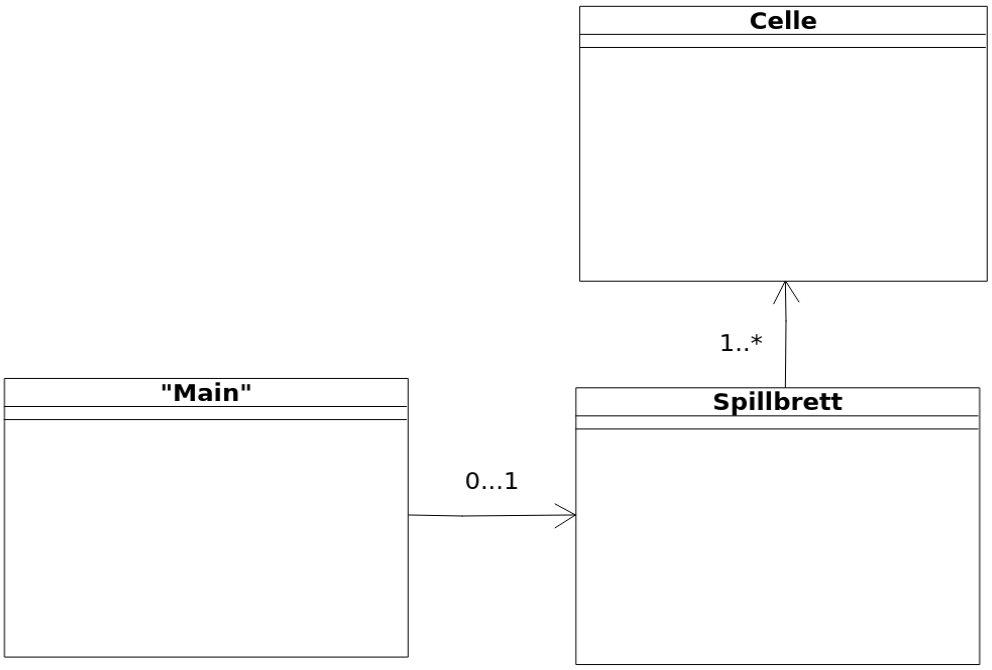


*Illustrasjon 2: To generasjoner av celler i et 7*7-størrelse spillebrett*

Struktur

Programmet består 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 *hentStatusTegn* 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.

4. Skriv en test funksjon *testCelle* som tester om *erLevende* og *hentStatusTegn* returnerer de riktige verdiene i følgende tre scenarier:
    a. med et nytt Celle-objekt (som skal oppføre seg som en død celle)
    b. med et Celle-objekt etter å ha kalt *settLevende*
    c. med et Celle-objekt etter å ha kalt settDoed
    d. For tips om hvordan du skriver test funksjoner, se gjennom undervisningsmodule Å skrive tester til funksjoner.

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 konstruktøren for klassen Spillebrett. Konstruktøren tar imot dimensjoner på spillebrettet og lagrer disse i instansvariablene *self._rader* og *self._kolonner*. Konstruktøren skal også opprette en variabel som holder styr på *generasjonsnummer* og som skal økes hver gang brettet oppdateres.

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

c. Metoden *_leggCellerIRutenett* som itererer over radene og kolonnene i rutenettet og erstatter None-verdiene med nye Celle-objekter.

Sørg for å kalle de riktige metodene inne i konstruktøren for å lage rutenettet
Kjør den ferdige funksjonen testLagRutenett so du finner i fila "Test_Spillebrett.py" som følger med denne øvingsoppgaven. Denne funksjonen skriver "Alt riktig" eller en beskjed om hvilken av metodene over som ikke gjør det den skal. Det kan også være veldig lurt å se på hvordan metodene over blir testet i den ferdige funksjonen testLagRutenett - dette kan hjelpe deg med å se hva disse metodene helt presist skal gjøre og returnere!

3. 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 ⅓ 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.*

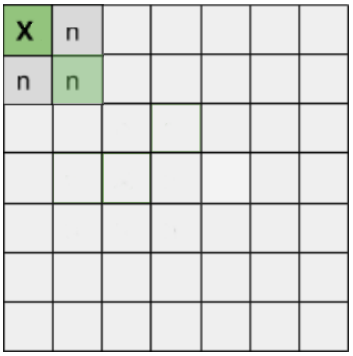
4. For å vise frem og teste spillebrettet skal du skrive metoden *tegnBrett*. Denne metoden skal bruke en nøstet for-løkke for å skrive ut hvert element i rutenettet. Husk å teste programmet ditt så langt ved å opprette et Spillbrett-objekt og skrive ut brettet i et lite testprogram. Dette kan du gjøre 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.

5. Skriv metoden *hentCelle* som tar imot en celles koordinater (rad og kolonne) i rutenettet, og returnere cellen til den gitte posisjonen. Hvis en ulovlig rad- eller kolonneindeks er gitt, returner None.

6. 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 og returnere en liste med alle cellens naboer. Husk at du kan bruke *hentCelle* til å hente celler uten å få feilmeldinger for ulovlige indekser. Det vil si, om du f.eks. kaller *hentCelle 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. Pass på kanter og hjørner.



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

7. 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 "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 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 testes ved å fylle rutenettet med et kjent mønster og se at det endrer seg som forventet etter en generasjon.

8. I tillegg til generasjonsnummer er vi interessert i den generelle cellestatusen på spillebrettet. Du skal derfor skrive en metode *antallLevende* 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.

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.

Eksempel på utskrift av spillebrettet

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 _lagTomRad(self):
        pass

    def _lagTomtRutenett(self):
        pass

    def _leggCellerIRutenett(self):
        pass

    def _generer(self):
        pass

    def tegnBrett(self):
        pass

    def oppdatering(self):
        pass

    def finnAntallLevende(self):
        pass

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

Main

Filnavn: main.py

```
from spillebrett import Spillebrett

def main():
    pass

# starte hovedprogrammet
main()
```

Test Spillebrett

Filnavn: Test_Spillebrett.py

```
from spillebrett import Spillebrett
from celle import Celle

def testKonstruktoer():
    testSpillebrett = Spillebrett(3, 5)

    for variabel in ["_rader", "_kolonner", "_generasjonsnummer"]:
        assert hasattr(testSpillebrett, variabel), f"manglende variabel " \
                                                   f"for Spillebrett: {variabel}"

    assert testSpillebrett._rader == 3, f"_rader var {testSpillebrett._rader}" \
                                        f" men det burde være 3"

    assert testSpillebrett._kolonner == 5, f"_kolonner var " \
                    f"{testSpillebrett._kolonner}, men det burde være 5"

    assert testSpillebrett._generasjonsnummer == 0, f"_generasjonsnummer var " \
                    f"{testSpillebrett._generasjonsnummer}, men det burde være 0"

def testLagTomRad(testSpillebrett):
    resultat = testSpillebrett._lagTomRad()
    beskjed = "forventet _lagTomRad() å returnere en rad av lengden 5 " \
              "(liste med 5 None verdier)"

    assert resultat is not None, f"{beskjed}, men ingenting ble returnert"

    assert type(resultat) == list, f"{beskjed}, men {resultat} ble returnert " \
                                   f"(typen er ikke list)"

    assert len(resultat) == 5, f"lengden på raden returnert av _lagTomRad()" \
                               f" var {len(resultat)}, men den skal være 5"

    assert resultat == [None]*5, f"{beskjed}, men {resultat} ble returnert"
```

(test kode fortsetter på neste side)

```
def testLagTomRutenett(testSpillebrett):
    resultat = testSpillebrett._lagTomtRutenett()
    beskjed = "forventet _lagTomtRutenett() å returnere en rutenett " \
              "(nøsted liste med None verdier)"

    assert resultat is not None, f"{beskjed}, men ingenting ble returnert"

    assert type(resultat) == list, f"{beskjed}, men {resultat} ble returnert " \
                                   f"(typen er ikke list)"

    assert len(resultat) == 3, f"rutenettet returnert av _lagTomtRutenett() " \
        f"skal inneholde 3 kolonner, men resultatet har lengde {len(resultat)}"

    for rad in resultat:
        assert type(rad) == list, f"{beskjed}, men typen av den indre " \
                                  f"elementer er ikke list"

        assert len(rad) == 5, f"Lengden på raden (indre liste) returnert " \
                              f"av _lagTomtRutenett() var {len(rad)}, men " \
                              f"den skal være 5"

        assert rad == [None]*5, f"{beskjed} med 3 rader og 5 kolonner, men " \
                                "f{resultat} ble returnert"

def testLeggCellerIRutenett(testSpillebrett):
    rutenett = testSpillebrett._rutenett
    for rad in rutenett:
        for verdi in rad:
            assert verdi is not None, "forventet at rutenettet ble fylt " \
                                      "med Celle-objekter, men funnet None"

def testLagRutenett():
    testSpillebrett = Spillebrett(3, 5)

    for metode in ["_lagTomRad", "_lagTomtRutenett", "_leggCellerIRutenett"]:
        assert hasattr(testSpillebrett, metode), f"manglende metode " \
                                                 f"for Spillebrett: {metode}()"

    testLagTomRad(testSpillebrett)

    testLagTomRutenett(testSpillebrett)
    testLeggCellerIRutenett(testSpillebrett)
```

(test kode fortsetter på neste side)

```
def testHentCelle():
    testSpillebrett = Spillebrett(3, 5)

    assert hasattr(testSpillebrett, "hentCelle"), f"manglende metode " \
                                        f"for Spillebrett: hentCelle()"

    testRutenett = [[Celle(), Celle(), Celle(), Celle(), Celle()],
                    [Celle(), Celle(), Celle(), Celle(), Celle()],
                    [Celle(), Celle(), Celle(), Celle(), Celle()]]

    testSpillebrett._rutenett = testRutenett

    for rad in range(3):
        for kolonne in range(5):
            resultat = testSpillebrett.hentCelle(rad, kolonne)
            assert isinstance(resultat, Celle), f"forvented at " \
                f"hentCelle({rad}, {kolonne}) returnerte et Celle-objekt," \
                f" funnet {type(resultat)}"

            assert resultat == testRutenett[rad][kolonne], "feil Celle " \
                f"objekt returnert av hentCelle({rad}, {kolonne}). Er du " \
                f"sikker på at indeksen til den returnerte cellen er riktig?"

    for rad in [-1, 4]:
        for kolonne in [-1, 6]:
            resultat = testSpillebrett.hentCelle(rad, kolonne)
            assert resultat is None, f"forventet at " \
                f"hentCelle({rad}, {kolonne}) returnerte None, " \
                f"funnet {type(resultat)}"
```

(test kode fortsetter på neste side)

```
def testFinnNabo():
    testSpillebrett = Spillebrett(2, 3)

    assert hasattr(testSpillebrett, "finnNabo"), f"manglende metode " \
                                        f"for Spillebrett: finnNabo()"


    testRutenett = [[Celle(), Celle(), Celle()],
                    [Celle(), Celle(), Celle()]]

    testSpillebrett._rutenett = testRutenett

    resultat = testSpillebrett.finnNabo(0, 0)

    assert resultat is not None, ".finnNabo(0, 0) returnerte ingenting"

    assert type(resultat) == list, f"forventet at .finnNabo(0, 0) returnerte " \
        f"en liste av Celle-objekter, men den returnerte {resultat}"

    assert None not in resultat, "listen returnert av finnNabo(0, 0) " \
        "inneholder None-verdier, men den skal bare inneholde Celle-objekter"

    assert testRutenett[0][0] not in resultat, "listen returnert av " \
        ".finnNabo(0, 0) inneholder Celle-objekten med index [0][0], " \
        "men den bør bare inneholde naboer av denne Celle."

    assert len(resultat) == len(set(resultat)) == 3, "forvented at listen " \
        "returnert av .finnNabo(0, 0) inneholder 3 unike nabo Celle-objekter," \
        f" funnet {len(set(resultat))}"

    naboer = [testRutenett[0][1], testRutenett[1][0], testRutenett[1][1]]

    assert set(resultat) == set(naboer), "listen returnert av .finnNabo(0, 0)" \
        " inneholder feil Celle-objekter. Er du sikker på at indeksen til " \
        "den returnerte cellen er riktig?"
```

(test kode fortsetter på neste side)

```
def testOppdatering():
    testSpillebrett = Spillebrett(4, 5)

    assert hasattr(testSpillebrett, "oppdatering"), f"manglende metode " \
                                        f"for Spillebrett: oppdatering()"

    testRutenett = [[Celle(), Celle(), Celle(), Celle(), Celle()],
                    [Celle(), Celle(), Celle(), Celle(), Celle()],
                    [Celle(), Celle(), Celle(), Celle(), Celle()],
                    [Celle(), Celle(), Celle(), Celle(), Celle()]]

    testRutenett[2][1].settLevende()
    testRutenett[2][2].settLevende()
    testRutenett[1][3].settLevende()

    testSpillebrett._rutenett = testRutenett

    testSpillebrett.oppdatering()

    levendeKoordinater = [(1,2), (2,2)]

    testSpillebrett.tegnBrett()

    for rad_idx in range(testSpillebrett._rader):
        for kol_idx in range(testSpillebrett._kolonner):

            skalVaereLevende = (rad_idx, kol_idx) in levendeKoordinater
            forvented, funnet = ("levende", "død") if skalVaereLevende \
                                else ("død", "levende")
            celle = testSpillebrett._rutenett[rad_idx][kol_idx]

            assert celle.erLevende() == skalVaereLevende, "Celle på rad " \
                f"{rad_idx} og kolonne {kol_idx} er {funnet}, men den " \
                f"skal vaere {forvented}"
```

(test kode fortsetter på neste side)

```
def testAntallLevende():
    testSpillebrett = Spillebrett(3, 5)

    assert hasattr(testSpillebrett, "antallLevende"), f"manglende metode " \
                                        f"for Spillebrett: antallLevende()"

    testRutenett = [[Celle(), Celle(), Celle(), Celle(), Celle()],
                    [Celle(), Celle(), Celle(), Celle(), Celle()],
                    [Celle(), Celle(), Celle(), Celle(), Celle()]]

    testRutenett[0][0].settLevende()
    testRutenett[1][1].settLevende()
    testRutenett[1][0].settLevende()
    testRutenett[1][2].settLevende()

    testSpillebrett._rutenett = testRutenett

    resultat = testSpillebrett.antallLevende()
    assert resultat is not None, ".antallLevende() returnerte ingenting"
    assert resultat == 4, "forventet antall levende " \
                f"celler i det oppgitte rutenettet til å være 4, " \
                f"men _antallLevende() returnerte {resultat}"


testKonstruktoer()
testLagRutenett()
testHentCelle()
testFinnNabo()
testOppdater()
testAntallLevende()
```