# Programmering og Modellering (PoM)
# Ugeseddel 6 — Uge 41 — Deadline 10/10

Kim Steenstrup Pedersen, Katrine Hommelhoff Jensen, Knud Henriksen,
Mossa Merhi og Hans Jacob T. Stephensen

30. september 2014

## 1 Plan for ugen

Efter at have stiftet bekendtskab med de forskellige elementer af programmeringssproget Python skal vi i denne uge anlægge nogle mere overordnede betragtninger og se, hvordan man kan bruge programmer til problemløsning. Mere præcist vil emnerne for tirsdagsforelæsningen være *programkonstruktion og programverifikation* og for torsdagsforelæsningen *indkøring og afprøvning* af programmer.

Under programudførelse kan systemfunktioner *kaste undtagelser*, hvis de for eksempel kaldes med parametre af forkert type, med meningsløs værdi eller i forkert antal, men vi skal nu se, at man også selv kan lade funktioner kaste undtagelser, og at man kan konstruere programmer til at *gribe undtagelser*, så de ikke nødvendigvis afbryder beregningsforløbet. Kast af brugerdefineret undtagelse (sprogelementet `raise`) diskuteres i denne uge, og det vises, hvordan eventuelle undtagelser kastet fra en indre programdel kan gribes (sprogelementerne `try:/except:`).

Tirsdagsforelæsningen kan siges at dreje sig om vekselvirkningen mellem programmets dynamiske sætninger på den ene side og beskrivelser (i form af logiske udsagn) af de statiske tilstande på den anden. Som støtte ved indkøring og afprøvning af programmer indeholder Python `assert`-sætninger og den særlige systemvariabel `__debug__`. Vi vil komme ind på specifikation af programmer og programdele med forudsætninger (*preconditions*) og krav (*postconditions*), på såkaldte øjebliksbilleder eller invarianter og på programkonstruktion ved trinvis forfining (*top down programming*) eller ved syntese (*bottom up programming*).

I yderste konsekvens kunne man forestille sig en egentlig formel verifikation af, at programmet opførte sig som ønsket, og opfattelsen af programmer som "prædikat-transformationer" vil kort blive præsenteret. Også objektorienteret problemanalyse og "programmering som teori-bygning" vil blive berørt.

Torsdag omtales intern og ekstern afprøvning, og der gives nogle tip om afprøvningsstøttet programmering, "unit testing" og "reviews". Desuden præsenteres den obligatoriske opgave (spillet *Life*).

**Til tirsdag:**

Læs Guttag kap. 6 og 7, samt side 91 fra Beazleys bog (findes under "Undervisningsmateriale" som "Python Essential Reference") samt beskrivelsen af programverifikation på adressen

`http://en.wikipedia.org/wiki/Hoare_logic`

**Til torsdag:**

Læs Steve McConnells kapitel: *Reviews in Software Quality Assurance* og Anders Borums noter (findes under "Undervisningsmateriale" som henholdsvis "Reviews" og "Afprøvning").

Læs også beskrivelsen af John Horton Conways spil *Life* på Wikipedia på adressen

`http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

### 1.1 Gruppeopgave

Den 10/10 senest klokken 15:00 skal besvarelse af følgende opgave afleveres elektronisk via Absalon. Opgaven skal besvares i grupper bestående af 1 til 3 personer og skal godkendes, for at

gruppedeltagerne kan kvalificere sig til den afsluttende tag-hjem eksamen. Opgavebesvarelsen skal oploades via kursushjemmesiden på Absalon (find underpunktet `ugeseddel6` under punktet `Ugesedler og opgaver`). Kildekodefiler ("script"-filer) skal afleveres som "ren tekst", sådan som den dannes af `emacs`, `gedit`, `Notepad`, `TextEdit` eller hvilket redigeringsprogram man nu bruger (*ikke* PDF eller HTML eller RTF eller MS Word-format). Filen skal navngives *efternavn1.efternavn2.efternavn3*.41.py, mens andre filer skal afleveres som en PDF-fil med navnet *efternavn1.efternavn2.efternavn3*.41.pdf.

6g1 I Scientific American for oktober 1970 beskrev den britiske matematiker John Horton Conway "spillet" Life, hvor kunstige organismer optager de kvadratiske celler i et ubegræset kvadratisk gitter. Spillet udvikler sig i skridt, man kunne kalde "generationer", idet hver kvadratisk celle i en bestemt generation befinder sig i en ud af to mulige tilstande, vi kan kalde "levende" eller "død".

Hver kvadratisk celle har otte naboer, nemlig de celler, med hvilke den har en kant eller et hjørne fælles, og cellens tilstand i en vis generation er entydigt bestemt af dens og dens naboers tilstand i foregående generation efter følgende regler:

- Hvis en levende celle har netop to eller tre levende naboer, overlever den til næste generation, og ellers dør den.
- Hvis en død celle har netop tre levende naboer, fødes den til live i næste generation, men ellers forbliver den død.

De beskrevne regler anvendes parallelt på alle celler, hvorved hver generation bestemmes af den foregående generation, og hele spillet entydigt udvikler sig med en startgeneration som kim.

Det viser sig, at disse regler er meget velvalgte, idet startgenerationer kan udvikle sig yderst forskelligt: Nogle vil være stabile eller periodiske, nogle uddør efter et vist antal generationer, og andre bevæger sig eller ekspanderer.

Se eventuelt mere på `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

**Overvejelser** Opgaven har følgende elementer, som leder frem til et system til visualisering af spillet Life:

1. Beslut en form, under hvilken en generation af spillet kan repræsenteres i Python. Det kunne for eksempel være som en liste af lister, en liste af koordinatpar, en hashtabel af koordinatpar eller som en værdi af typen `set` (se nærmere i dokumentationen af Python om `set`).
2. Af praktiske grunde lader det sig kun gøre at vise et rektangulært udsnit af det principielt uendelige kvadratgitter. Det er derfor nødvendigt at beslutte, hvorledes celler skal opføre sig ved randen af det rektangulære udsnit. Her er nogle forslag:
   (a) Man kunne vælge kun at lade reglerne gælde inden for udsnittet (så der aldrig blev født celler udenfor).
   (b) Man kunne lægge udsnittet omkring de levende celler, så rektanglet skrumpede eller (i det omfang, der var plads på skærmen) udvidede sig efter behov. Måske skulle rektanglets grænser så indgå som komponent i repræsentationen af en generation.
   (c) Man kunne identificere modstående kanter, svarende til, at spillet foregik på en torus (ring). Hvis rektanglet havde $m$ rækker og $n$ søjler, nummereret henholdsvis $0, 1, \ldots, m-1$ og $0, 1, \ldots, n-1$, svarer det beregningsmæssigt til, at man placerer en celle, der logisk hørte hjemme på plads `(`$i$`,`$j$`)`, inden for rektanglet på den faktiske position `(`$i$ `%` $m$`,`$j$ `%` $n$`)`.

**Hjælpefunktion** For at gøre det let for kursusdeltagerne at fremstille udviklingen under et spil Life grafisk stiller vi funktionen `visLife` til rådighed. Dette er en højereordensfunktion, som skal kaldes på formen

```
visLife(foerste,naeste,levende)
```

idet de tre argumenter `foerste`, `naeste` og `levende` skal være funktioner, der kan kaldes på følgende måde:

`foerste()` Et sådant kald (uden parametre) skal beregne spillets startgeneration og som funktionsværdi returnere et kvadrupel $(i_{min}, i_{max}, j_{min}, j_{max})$, der markerer udstrækningen af det rektangel, inden for hvilket de levende celler skal vises. Rektanglet kommer til at bestå af kvadrater med koordinater $(i,j)$, hvor $i_{min} \leq i \leq i_{max}$ og $j_{min} \leq j \leq j_{max}$.

`naeste()` Et sådant kald (uden parametre) skal beregne næste generation og som funktionsværdi returnere et kvadrupel $(i_{min}, i_{max}, j_{min}, j_{max})$, der på samme måde som for `foerste` markerer udstrækningen af det rektangel, inden for hvilket de levende celler skal vises.

`levende(`$i$`,`$j$`)` Idet $i$ og $j$ skal have heltallig type, skal returværdien være `True`, hvis cellen på plads $(i,j)$ er levende, og ellers `False`.

Funktionskaldet `visLife(foerste,naeste,levende)` åbner på skærmen et vindue (*canvas*), i hvilket spillet kan udvikle sig, og derefter gælder følgende:

- Kvadratnettets udstrækning bestemmes ved hjælp af et indledende funktionskald

  $(i_{min}, i_{max}, j_{min}, j_{max})$ `= foerste()`

  hvorefter gitteret tegnes. Desuden tegnes den første generation, idet cellers status afgøres af funktionskald

  `levende(`$i$`,`$j$`)`

  for alle $i_{min} \leq i \leq i_{max}$ og $j_{min} \leq j \leq j_{max}$.
- Funktionen `visLife` animere spillets generationer i diskrete skridt ved for hvert skridt at kalde `naeste()`. Dette kald medfører at `visLife` tegner næste generation, idet status for celler inden for rektanglet afgøres af kald af formen `levende(`$i$`,`$j$`)` på samme måde som ved startgenerationen.
- Ved at lukke tegnevinduet vil kaldet af `visLife` afslutte og få tegnevinduet til at forsvinde.

[Note: Funktionen `visLife` er implementeret som en klasse og det er konstruktørmetoden vi kalder — mere om det i uge 47 - 48. Her kan vi blot betragte `visLife` som en hvilken som helst anden funktion.]

**Program** Opgaven besvares med et `python`-program, som indledes med kommentarer eller tekststrenge, der gør rede for de trufne beslutninger med hensyn til repræsentation og udsnitsrektangel.

Derefter følger de definitioner og størrelser, der skal benyttes til repræsentation af en generation af spillet, og de tre beskrevne funktioner `foerste`, `naeste` og `levende` defineres.

Selv om disse funktioner ikke eksplicit har repræsentation af spillet som en af deres funktionsparametre, er det klart, at deres funktionskroppe må afhænge af denne repræsentation.

Husk, at hvis en funktion ikke kun konsulterer en udefrakommende global variabel, men også har brug for at ændre den, kan det være nødvendigt med en `global` erklæring.

Funktionen `visLife` er indeholdt i filen `kursusuge6modul.py`, der ligger på kursets Absalonside under Ugesedler og opgaver → Ugeseddel 6. Hent denne fil ned til din egen computer, og placer den i det samme katalog som det, hvori programmet til besvarelse af opgaven ligger. Efter programlinjen

```
from kursusuge6modul import *
```

vil `visLife` være til rådighed. (Første gang, modulet benyttes, sørger systemet for, at det bliver oversat, og filen `kursusuge6modul.pyc` med det oversatte program vil også dukke op i kataloget.) Opgaveløsere kan modificere modulet efter forgodtbefindende; det vil for eksempel være enkelt at ændre cellernes størrelse eller valget af farver.

Ved vurdering af opgavebesvarelser vil der blive lagt vægt på en omhyggelig og systematisk afprøvning. Vi anbefaler, at hver af de tre hjælpefunktioner afprøves grundigt hver for sig (det, som kaldes *unit test*), inden de forsøges indsat i `visLife`.

Ved kørsel af opgavebesvarelsens program skal spillet Life starte med følgende konfiguration af levende celler:

```
(10,6), (10,7), (10,8), (9,8), (8,7)
```

Det er tilladt at inkludere yderligere start konfigurationer i besvarelsen. Dette kan eksempelvis gøres ved at bede brugeren vælge mellem konfigurationer ved indtastning på tastaturet.

## 1.2 Tirsdagsøvelser

Besvarelser af disse opgaver skal ikke afleveres, men opgaverne forventes løst inden torsdag.

6ti1 Definer en funktion `graenser(`$xys$`)`, der med en liste af talpar `[(`$x_1$`,`$y_1$`), (`$x_2$`,`$y_2$`), ..., (`$x_n$`,`$y_n$`)]` som argument returnerer et kvadrupel `(`$xmin$`,`$xmax$`,`$ymin$`,`$ymax$`)` med de mindste og de største forekommende værdier af $x$ og $y$. (Formelt: For alle $i = 1, 2, \ldots, n$ skal $xmin \leq x_i \leq xmax$ og $ymin \leq y_i \leq ymax$, og hvert af de fire lighedstegn skal antages for mindst én værdi af $i$.)

Hvis listen er tom ($n = 0$), skal funktionen returnere kvadruplet `(1,0,1,0)`.

6ti2 Til administration af mængder, hvis indhold ændrer sig dynamisk under programudførelse, defineres fire funktioner med følgende virkning:

`empty()`: Returnerer (en repræsentation af) den tomme mængde $\emptyset$.

`insert(`$g$`,`$x`)`: Skal ændre $g$ til at repræsentere $M \cup \{x\}$, hvis $g$ tidligere repræsenterede $M$.

`member(`$g$`,`$x`)`: Returnerer `True` eller `False` efter som $x \in M$ eller $x \notin M$, hvor $g$ repræsenterer $M$.

`toList(`$g$`)`: Returnerer en liste bestående af elementerne i (mængden repræsenteret af) $g$.

OBS.: Et forslag til, hvorledes disse funktioner kunne implementeres, stilles sidst i denne opgaveformulering, men i første omgang er det meningen, at man kun skal bygge på beskrivelsen ovenfor og ikke udnytte nogen konkret implementering.

Definer funktionen `fromList(`$zr$`)`, der kaldt med en liste `[`$z_1$`, `$z_2$`, ..., `$z_n$`]` af elementer som argument returnerer (en repræsentation af) mængden bestående af disse elementer. Funktionskroppen skal benytte de ovennævnte funktioner `empty()` og `insert()` uden at bygge på, hvordan de er implementeret.

En naiv implementering af funktionerne:

```
def empty():
    return []
def member(g,x):
    return x in g
def insert(g,x):
    if not member(g,x):
        g.append(x)
def toList(g):
    return g
```

6ti3 I denne opgave betragtes igen de fire funktioner `empty()`, `insert()`, `member()` og `toList()` fra 6ti2, men for at undgå de langsommelige lineære listegennemløb besluttes det at bruge en hashtabel snarere end en liste som mængderepræsentation. Mere præcist skal hashtabellens nøgler netop være mængdens elementer, mens tabellens værdier simpelt hen for samtlige opslag skal være `True`.

Funktionen `insert()` programmeres nu for eksempel

```
def insert(g,x):
    g[x] = True
```

Programmer de øvrige tre funktioner.

## 1.3 Torsdagsøvelser

Besvarelser af disse opgaver skal ikke afleveres, men opgaverne forventes løst inden tirsdag i efterfølgende uge.

6to1 Ud fra et ønske om også at kunne fjerne elementer fra de mængder, der opereres på, specificeres funktionen

`delete(`$g$`,`$x$`):` Skal ændre $g$ til at repræsentere $M \setminus \{x\}$, hvis $g$ tidligere repræsenterede $M$.

Giv både en `python`-definition af `delete()` for den naive mængdeimplementering foreslået i opgave 6ti2 og for implementeringen som hashtabel fra opgave 6ti3.

6to2 Konstruer afprøvningsdata for funktionerne `empty()`, `insert()`, `delete()`, `member()` og `toList()` fra opgaverne 6ti2 og 6to1, og afprøv implementeringen af mængder som hashtabeller (foreslået i opgave 6ti3 og 6to1).

6to3 Konstruer afprøvningsdata for funktionen `graenser()` fra opgave 6ti1, og afprøv den.

6to4 En generation under spillet *Life* repræsenteres som mængden af levende celler. Intentionen med nedenstående funktion `naboer()` er, at den skal optælle antallet af levende celler i $3 \times 3$-kvadratet omkring et givet punkt, men den her viste definition er behæftet med en række fejl. Find og ret disse fejl. (Funktionen `member()` er defineret i opgave 6ti2 eller 6ti3.)

```
def naboer(g,x,y):
    n = 0
    for x-1 <= u <= x+1:
        for v in range(y-1,y+1):
            if member(g,(x,y)):
                n = n + 1
    return n
```

6to5 En generation under spillet *Life* repræsenteres som mængden af levende celler, funktionerne `empty()`, `insert()`, `member()` og `toList()` er defineret i opgave 6ti2 eller 6ti3, `graenser()` i opgave 6ti1, `naboer()` i opgave 6to4 og `fromList()` i opgave 6ti2.

Brug de nævnte funktioner til at definere funktionen `next(`$g$`)`, som returnerer næste generation af spillet efter generationen $g$.

Betragt startkonfigurationen

```
glider = fromList([(0,1), (1,2), (2,0), (2,1), (2,2)])
```

Udregn i hånden på et ark kvadreret papir et par af de efterfølgende generationer, og test den konstruerede funktion med kald som for eksempel

```
toList(next(glider))
```

og

```
toList(next(next(glider)))
```

For bedre at få indtryk af bevægelsen kan man føje en stabil blok som for eksempel `[(-1,-2), (-1,-3), (-2,-2), (-2,-3)]` til startkonfigurationen.