Programmering og Modellering (PoM)
Ugeseddel 10 — Uge 47 — Deadline 28/11

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

6. november 2014

1 Plan for ugen

Denne uge tager vi fat på et nyt emne: Objektorienteret programmering. Indtil videre har vi brugt indbyggede typer, såsom heltal og tekststrenge, og vi har gjort god brug af de tilhørende funktioner og operatorer. I denne uge skal vi lære at lave vores egne *abstrakte datatyper*. Nøgleord for denne uge vil være: *abstrakte datatyper*, *klasser*, *instanser*, *attributter*, *metoder* med særlig fokus på `__init__` og `__str__`, funktioner med og uden *sideeffekt*, *objektorienteret programmering*, samt hvorledes man ændre på betydningen af *operatorer* (*Eng. operator overloading*).

Til tirsdag:
Læs Guttag kapitel 8.1 og 8.3.

Til torsdag:
Læs Guttag kapitel 8.1 og 8.3.

Bemærk: Du har to uger til at løse den individuelle obligatoriske opgave og der er nye lokaler til øvelsesundervisningen (se Kursusoversigt i Absalon for detaljer).

1.1 Individuel opgave

Den 28/11 senest klokken 15:00 skal besvarelse af følgende opgave afleveres elektronisk via Absalon. Opgaven skal besvares individuelt og skal godkendes, for at du kan kvalificere dig til den afsluttende tag-hjem eksamen. Opgavebesvarelsen skal oploades via kursushjemmesiden på Absalon (find underpunktet `ugeseddel10` 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 *fornavn.efternavn*.47.py, mens andre filer skal afleveres som en PDF-fil med navnet *fornavn.efternavn*.47.pdf.

10i1 Lineær algebra handler om koblede lineære ligninger, såsom

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = y_1, \tag{1a}$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = y_2, \tag{1b}$$
$$\vdots$$
$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = y_m, \tag{1c}$$

hvor a_{ij} er koefficienterne, der relaterer variablene x_j og y_i. Dette skrives ofte på kort form,

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & & \ddots & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}, \tag{2}$$

eller endnu kortere

$$\mathbf{A}\vec{x} = \vec{y}, \tag{3}$$

hvor $\mathbf{A}$ er en matrix og $\vec{x}$ og $\vec{y}$ er (søjle) vektorer. Håndtering af lineære ligningssystemer er et centralt element i mange videnskabelige problemer, dels fordi de nemmere kan behandles end systemer af højereordens polynomier, og fordi mange fænomener kan modelleres som et eller flere lineære ligningssystemer. Reglerne, for hvordan man regner med lineære ligningssystemer på matrixform, kaldes Lineær Algebra som dækkes af kurset Lineær Algebra (blok 2), men man kan allerede nu læse mere derom bla. på `http://da.wikipedia.org/wiki/Matrix`. Der findes mange vektor-matrix- biblioteker til Python, men denne opgave går ud på at programmere dele af et vektor-matrix-bibliotek fra bunden samt at udføre en egentlig afprøvning af biblioteket. Der må kun benyttes de indbyggede Python 2.7-kommandoer:

a: Definer klassen `simplematrix`

Definer en klasse `simplematrix`. Klassen skal kunne repræsentere en matrix (tabel) med $m > 0$ rækker og $n > 0$ søjler hvis elementer er reelle tal, samt den tomme 0×0 matrix med $m = n = 0$. Klassen skal som minimum have metoderne:

`__init__`,
: som initialiserer matricens værdier. Funktionen skal

  i. uden argumenter oprette en 0×0 matrix (den tomme matrix),

  ii. med præcis 2 argumenter m, n oprette en matrix med m rækker og n søjler, hvis elementer er sat til 0, og endeligt

  iii. med 3 argumenter $m, n, values$ oprette en $m \times n$ matrix, hvis værdier kopieres fra sekvensen $values$, således at første element i values sættes på matrixplads (1,1), andet på matrixplads (1,2) osv. (dvs. række først orden).

`__str__`,
: som returnerer en tegnstreng, der opstiller matricens værdier på tabelform (række for række med linjeskift efter hver række).

`read(filename)`,
: som erstatter matricen med den, som indlæses fra filen `filename`. Valg af filformat er en del af opgaven.

`write(filename)`,
: som udskriver matricens værdier til filen `filename` på en måde, som kan læses af `read(filename)`.

samt operatorerne

`__add__(self, other)`,
: som elementvis adderer 2 matricer af samme størrelse.

`__mul__(self, other)`,
: som matrix-multiplicerer 2 matricer når antallet af søjler i den første er lig antallet af rækker i den næste.

`__eq__(self, other)`,
: som elementvis sammenligner 2 matricer af samme dimension og returnerer `True`, hvis og kun hvis alle elementer er ens, ellers returneres `False`. Denne metode overskriver sammenligningsoperatoren `==`.

`__ne__(self, other)`,
: som elementvis sammenligner 2 matricer af samme dimension og returnerer `False`, hvis og kun hvis alle elementer er ens, ellers returneres `True`. Denne metode overskriver sammenligningsoperatoren `!=`.

Koden skal naturligvis indeholde passende docstrings såvel som andre kommentarer der dokumentere din løsning.

b: Afprøv klassen `simplematrix`

For at sikre kvaliteten af et givent program er det nødvendigt at afprøve det. Afprøvning er lidt af en kunst og især af et program, man selv har skrevet, da det kan være svært at opnå en passende mental distance til programmet. Denne opgave går ud på at designe en grundig afprøvning, hvor alle relevante afprøvningsforsøg identificeres og programmeres. Afprøvningen dokumenteres med et program, hvor et antal afprøvningsforsøg er programmeret efter følgende skabelon:

```
print "Nu afprøves matrixaddition for tilfælde 2 lige store matricer"
A = simplematrix(2 ,2, [1,2,3,4])
B = simplematrix(2, 2, [5,6,7,8])
C = simplematrix(2, 2, [6,8,10,12])
print C == A + B
```

1.2 Tirsdagsøvelser

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

10ti1 Vi har allerede uden at vide det arbejdet med objekter i Python. Datatyper som `String`, `List` og `Dict` er alle eksempler på abtrakte datatyper og når vi opretter (også kaldet *instantiering*) variable af disse typer så har vi skabt det vi kalder objekter. I Python kan vi definere abstrakte datatyper ved hjælp af klasser hvori vi definere datastrukturer og funktioner på disse - vi kalder sådanne funktioner for metoder og de manipulere objekter som vi instantiere fra klassen. På lister har vi allerede kaldt metoder på objekter - eksempelvis `L=[1, 2]; L.append(3)`. Prøv at oprette følgende klasse definition:

```
class B(object):
   """En klasse"""
   pass
```

Du kan instantiere objekter fra klassen `B` ved at skrive `b = B()`. Prøv at benytte `type` funktionen på både `b` og `B` og se hvilken type disse udtryk har.

10ti2 Som første eksempel på en abstrakt datatype, lad os definere en klasse `Point` som er en skabelon til at kunne repræsentere et punkt i et 2-dimensionelt rum:

```
class Point(object):
       """Represents a point in 2-D space.
       Attributes: x, y"""
```

Vi kan oprette et punkt, give den to attributter `x, y` og tilgå disse således:

```
p = Point()
p.x = 3
p.y = 4
print p.x, p.y
```

(Vi skal senere se smartere versioner af klassen `Point` som ikke tilføjer attributer til objektet efter instantiering som ovenfor.)

Skriv en funktion `distance_between_points` som tager to punkter af typen `Point` som parametre og returnere den (Euklidiske) afstand mellem punkterne.

10ti3 Lad os definere en klasse `Rectangle` som skal kunne repræsentere et rektangel i et 2-dimensionelt rum

```
class Rectangle(object):
        """Represents a rectangle.
        Attributes: width, height, corner
        """
```

Repræsentation består af et punkt som repræsentere nederste venstre hjørne af rektanglet (som vi vil repræsentere ved hjælp af `Point` klassen fra opgave 10ti2) samt attributter som repræsentere bredde og højde af rektanglet. Således

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

Skriv en funktion `move_rectangle(rec, dx, dy)` som tager en instans `rec` af klassen `Point` som parameter samt parametrene og tallene `dx, dy`, og flytter rektanglet ved at lægge `dx` hhv. `dy` til `rec.corner` attributens `x` hhv. `y` attributter. Funktionen skal ændre `rec` (ikke returnere et nyt rektangel).

1.3 Torsdagsøvelser

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

10to1 Skriv en ny udgave af klassen `Point` fra opgave 10ti2 som indeholder en konstruktørmetode `__init__` som tager to parametre `x, y` og opretter tilsvarende attributer i instansen af klassen. Er det nødvendigt at omskrive din implementation af funktionen `distance_between_points` for at kunne anvende denne nye forbedrede udgave af klassen `Point`?

10to2 Skriv en `__str__` metode til klassen `Point` fra opgave 10to1. Lav en instant af klassen `Point` og udskriv objektet vhj. `print`.

Skriv en afprøvning af metoden. Afprøvningen skal designes således, at alle skrevne programlinjer bliver afprøvet (også kaldet en *intern afprøvning*), samt at specifikationens grænsetilfælde afprøves (også kaldet en *ekstern afprøvning*).

10to3 Når vi laver vores egne abstrakte datatyper i Python skal vi passe på når vi ønsker at kopiere objekter. Python giver os mulighed for to typer af kopiering - overfladisk kopi (*Eng. shallow copy*) og dyb kopi (*Eng. deep copy*). Forskellen mellem disse omtales i forelæsningerne.

Skriv en ny udgave af funktionen `move_rectangle` fra opgave 10ti3 som opretter og returnere en ny instans af `Rectangle` istedet for at modificere parameterobjektet `rec`.

10to4 Skriv en `__add__` metode til klassen `Point` fra opgave 10to1 som lægger et punkt til det punkt vi kalder metoden på. Metoden skal returnere et nyt objekt af typen `Point` med resultatet af summen.

Skriv en afprøvning af metoden. Afprøvningen skal designes således, at alle skrevne programlinjer bliver afprøvet (også kaldet en *intern afprøvning*), samt at specifikationens grænsetilfælde afprøves (også kaldet en *ekstern afprøvning*).

10to5 Skriv en ny udgave af klassen `Rectangle` fra opgave 10ti3 som indeholder en konstruktørmetode `__init__` som tager et passende antal parametre. Derudover skal klassen indeholde metoden `move_rectangle` som skal fungere som beskrevet i opgave 10to3. Kan du genbruge din implementation af funktionen fra opgave 10to3?

10to6 Skriv en `__eq__` metode til klassen `Point` fra opgave 10to1 som implementerer `==` operatoren for objekter af typen `Point`, dvs. at den skal returnere `True` hvis og kun hvis at koordinaterne i de to punkter vi sammenligner er identiske.

10to7 Udvid metoden `__add__` fra opgave 10to4 således at den enten kan tage et `Point` objekt eller en tuple som parameter.

1. Hvis parameteren er af typen `Point`, så skal metoden returnere et nyt `Point` objekt som indeholder summen af det `Point` objekt vi kalder metoden på og parameter objektet.
2. Hvis parameteren er af typen `tuple`, så skal metoden returnere et nyt `Point` objekt som indeholder summen af det `Point` objekt vi kalder metoden på og parameter objektet, således at første tuple værdi skal lægges til `Point.x` og anden tuple værdi skal lægges til `Point.y`.

Hvad sker der hvis en parameterværdi af typen `tuple` har antal elementer $n \neq 2$?

10to8 Skriv en afprøvning af metoden `__add__` fra opgave 10to7. Afprøvningen skal designes således, at alle skrevne programlinjer bliver afprøvet (også kaldet en *intern afprøvning*), samt at specifikationens grænsetilfælde afprøves (også kaldet en *ekstern afprøvning*).

10to9 Kan du udvide `__eq__` metode på klassen `Point` fra opgave 10to6 til også at kunne håndtere parametre af typen `tuple`?

10to10 Metoden `__add__` fra opgave 10to7 fungerer kun hvis vi har rækkefølgen `a+b` hvor `a` er en instans af type `Point` og `b` er en instans af type `Point` eller `tuple`. Hvis vi derimod forsøger os med at `a` er af typen `tuple`, så fejler metoden. Vi kan løse dette problem ved at tilføje metoden `__radd__` således at den enten kan tage et `Point` objekt eller en tuple som parameter.

1. Hvis parameteren er af typen `Point`, så skal metoden returnere et nyt `Point` objekt som indeholder summen af det `Point` objekt vi kalder metoden på og parameter objektet.
2. Hvis parameteren er af typen `tuple`, så skal metoden returnere et nyt `Point` objekt som indeholder summen af det `Point` objekt vi kalder metoden på og parameter objektet, således at første tuple værdi skal lægges til `Point.x` og anden tuple værdi skal lægges til `Point.y`.

Kan du anvende din implementation af `__add__` fra opgave 10to9 til at skrive `__radd__` metoden?