Oppgave fra oblig 3

Forestill deg at beboerne på et sykehjem får servert tre måltider i løpet av en dag. Du skal organisere informasjonen om fem beboere og hvilke måltider de får til frokost, lunsj og middag.

Lag en datastruktur som kobler hver beboer til en oversikt over hva vedkommende spiser til de tre måltidene. For eksempel: en person kan få brød til frokost, egg til lunsj og pølser til middag. Lag så en prosedyre som skriver ut navnene til alle beboerne og spør brukeren om å skrive navnet til en beboer i terminalen. Deretter skal prosedyren skrive ut matplanen til den valgte beboeren. Skriv ut en melding til brukeren hvis beboeren ikke er registrert.

Løs oppgaven ved å lage en Person-klasse

Mulige utvidelser

1. Lag en klasse for måltid, som holder styr på rettens navn, om det er ment for frokost, lunsj eller middag.
    a. Legg til en instansparameter *allergener*, på måltid og person-klassen. Lag så en metode i klassen Person som sjekker om de er allergiske mot noen av de planlagte måltidene.
2. Lag en klasse *Sykehjem* som holder en samling av *Personer* og som tilbyr metoder for å legge til, fjerne, og søke på personer i samlingen. Samt har en metode for å skrive ut en fin oversikt over alle beboerne på sykehjemmet.

Buss oppgave

I denne oppgaven skal du lage en klasse `Buss` for å modellere en busstur og passasjerer.

Et `Buss`-objekt har en makskapsitet som er antall mulige sitte- og ståplasser på bussen. I tillegg kan et `Buss`-objekt (1) plukke opp passasjerer *én etter én* hvis bussen ikke er full og (2) slippe av passasjerer *én etter én* hvis bussen ikke er tom.

a) Skriv klassen `Buss` med konstruktør og tilhørende metoder. Makskapasiteten skal tas inn som argument når vi oppretter et `Buss`-objekt.

b) Utvid klassen med metodene `er_tom()` og `er_full()` som skal returnere `True` eller `False` avhengig av om bussen er full eller ikke.

c) Skriv så metodene `plukk_opp()` og `slipp_av()` som sjekker om bussen er tom eller full og så henholdsvis øker eller reduserer antall passasjerer på bussen. Funksjonene skal også skrive ut en passende melding om hendelsen.

d) Lag til slutt en metode `hent_antall_passasjerer()` som returnerer antall passasjerer på bussen.

e) I *test_buss.py* lag en prosedyre `hovedprogram` som oppretter en instans av klassen med plass til 20 passasjerer *Husk å importere klassen.* Deretter simuleres følgende busstur:
- Plukk opp 10 passasjerer
- Plukk opp 12 passasjerer til
- Slipp av 18 passasjerer
- Slipp av 3 passasjerer

Skriv også ut antall passasjerer på bussen etter hver endring for å sjekke om det fungerer som forventet.

Pass også på at det skrives ut "Bussen er full!" om programmet prøver å plukke opp passasjerer når `Buss`-objektet er fullt og "Det er ingen passasjerer å slippe av!" om det prøver å slippe av passasjerer når `Buss`-objektet er tomt.

Mastermind-spillet

I denne oppgaven skal du lage en enkel versjon av MasterMind-spillet.

a) Skriv en klasse `MasterMind`. Et `MasterMind`-objekt skal ta vare på en kode som opprettes når objektet opprettes. Koden skal være en liste som består av fire tilfeldige heltall (hint: bruk `random.randint`).

b) Utvid klassen med en metode `hent_bruker_gjett()` som ber brukeren om en kode (en streng som består av fire heltall, f.eks. "1234"). Metoden skal konvertere denne strengen til en liste med fire heltall og returnere denne listen.

c) Lag en metode `sjekk_gjett()` som tar en gjetteliste med fire tall, sammenligner den med den riktige kodelisten og lager en liste med informasjon om gjettet. Listen skal inneholde tallet hvis det var gjettet riktig og "X" hvis tallet var feil.

F.eks. hvis koden er `[1, 2, 3, 4]` og gjettelisten er `[1, 2, 1, 2]`, skal metoden returnere `[1, 2, "X", "X"]`.

d) Lag en metode `spill()` som simulerer spillet. Metoden skal kalle på `hent_bruker_gjett()` og `sjekk_gjett()` i en løkke og kjøre så lenge brukeren ikke har gjettet riktig kode. Etter hvert gjett skal det skrives ut resultatet fra `sjekk_gjett()`. Når brukeren vinner, skrives det ut hvor mange ganger de gjettet en kode.

Utfordring (Tidligere eksamen)

I denne oppgaven skal du lage en forenklet utgave av spillet tre-på-rad, også kalt bondesjakk. Du skal gjøre dette ved å benytte objektorienterte prinsipper, altså med klasser og objekter.

Tre-på-rad, eller bondesjakk, foregår på et rutenett av 3*3 ruter. To spillere konkurrerer, og i hvert trekk får de plassere en brikke på rutenettet. Vinneren av spillet er den første spilleren som har tre brikker på rad, enten vertikalt eller horisontalt. I denne forenklede versjonen av spillet tar vi ikke høyde for tre på rad diagonalt.

Deloppgave 1

Lag en klasse *Spiller*. En spiller skal representeres på brettet enten med symbolet X eller symbolet O. Ved opprettelse gis spillerens symbol som parameter til konstruktøren. Om symbolet som gis ikke er X eller O skal det printes en beskjed om at symbolet er ugyldig.

Deloppgave 2

Lag en klasse *Rute*. En rute representerer en enkelt rute på et tre-på-rad-brett. Ruter er enten ledige eller opptatt, og om de er opptatt har de en referanse til spilleren som har plassert brikken sin der. Når en rute opprettes settes denne referansen til *None*.

Ruter har en instansvariabel *spiller* som starter som *None*.

Ruter har en metode *plasser_brikke(self, spiller)* som setter *spiller*-referansen til spilleren som gis som parameter.

Ruter har en metode *er_opptatt(self)* som returnerer *True* dersom ruten er tatt av en spiller, ellers *False*.

Ruter har en metode *hent_brikkeeier(self)*. Denne returnerer spiller-objektet som eventuelt ligger i spiller-referansen, ellers *None*.

Deloppgave 3

Lag klassen *Brett*. Et brett er et rutenett på 3*3 ruter, hvor hver rute representeres ved et objekt av klassen *Rute*. Datastrukturen er en nøstet liste, og rute-objekter opprettes i konstruktøren. Konstruktøren oppretter også to referanser til spiller-objekter som i utgangspunktet settes til *None*.

Skriv en metode *legg_til_spiller(self, symbol)*. Denne oppretter et *Spiller*-objekt med symbolet gitt som parameter, og setter inn dette objektet i en ledig spiller-referanse. Om ingen spiller-referanser er ledig, printes en beskjed om at det allerede eksisterer to spillere.

Skriv en metode *plasser_brikke(self, spiller, x, y)*. Spilleren gitt i parameteren forsøker å plassere en brikke i ruten på plass [x][y]. Sjekk om det er mulig å plassere en brikke på denne plassen, og send i så fall korrekt spiller-referanse til ruten i posisjonen og return True. Om ruten allerede er tatt, printes en beskjed om dette og det returneres False.