Exempeltentamen - TDDE24

Uppgift 1

Skriv en funktion liuid som tar ett förnamn och ett efternamn och returnerar ett lämpligt LiU-id. Ett sådant ska bestå av de tre första respektive de två första bokstäverna ur förnamnet och efternamnet, samt ett slumpmässigt tal i intervallet 100 till 999 (åtminstone i den här uppgiften). Om det inte finns tillräckligt många tecken i för- eller efternamnet så används de som finns. Användarnamnet ska enbart bestå av små bokstäver.

```
>>> liuid("Ture", "Teknolog")
'turte155'
>>> liuid("Bo", "Ek")
'boek675'
```

Tänk på att funktionen ska *returnera* korrekta värden:

```
>>> assert liuid("Ture", "Teknolog") == 'turte155'
>>> assert liuid("Bo", "Ek") == 'boek675'
```

För att kunna generera slumptal måste vi först importera en modul random. Därefter kan vi få fram slumpmässiga heltal med hjälp av funktionen randint. Se följande exempel där slumptal i intervallet 10-19 genereras:

```
>>> from random import randint
>>> randint(10, 19)
15
>>> randint(10, 19)
12
```

Glöm inte docstrings – detta är den sista påminnelsen.

Uppgift 2

Skriv en funktion replace som i en rak lista ersätter alla förekomster av ett element med ett annat. Funktionen ska finnas i två versioner: en rekursiv (replace_r) och en iterativ (replace_i). För båda versionerna gäller att listan endast får bearbetas en gång. Detta innebär bl.a. att inbyggda metoder som remove() inte kan användas, utan att listorna måste gås igenom element för element och nya listor byggas upp och returneras. Exempel:

```
>>> replace('old', 'new', ['old', 'a', 'old', 'b'])
['new', 'a', 'new', 'b']
>>> replace(1, 'x', [0, 1, 2, 1, 7])
[0, 'x', 2, 'x', 7]
```

Uppgift 3

Deluppgift 3A (3p)

Skriv en högre ordningens funktion collector som tar en predikatsfunktion och en lista. Funktionen collector ska returnera en lista med alla element, oavsett nivå, som uppfyller predikatsfunktionen. Elementen i resultatlistan ska komma i samma ordning som i ursprungslistan, men resultatlistan ska vara platt och inte innehålla några undernivåer. Funktionen får inte använda den inbyggda funktionen filter, utan måste gå igenom listan element för element. Exempel:

```
>>> collector((lambda x: isinstance(x, str)), ['a', 1, ['b', ['c'], 2], 'd', 3])
['a', 'b', 'c', 'd']
```

Deluppgift 3B (2p)

Skriv en funktion numbers_in_interval som med hjälp av collector plockar ut alla heltal som ligger inom ett angivet intervall (inklusive gränserna). Listan kan innehålla även andra saker än tal. Exempel:

```
>>> numbers_in_interval(10, 20, [['x', 5, [15, 'y'], 20], 2, 'z', 17])
[15, 20, 17]
```

Uppgift 4

I denna uppgift ska vi hantera ett program som spelar Tic-tac-toe, Tre-i-rad eller vad man nu väljer att kalla det. Det är i grund och botten en förenklad version av luffarschack. Man har en spelplan om 3 x 3 rutor. En spelare har ringar och en spelare har kryss. Spelarna turas om att lägga ut sina spelpjäser, och den första som får tre i rad (vågrätt, lodrätt eller diagonalt) vinner. Det finns också en möjlighet att det blir oavgjort om hela spelplanen fylls utan att någon har tre i rad.

I filen tictactoe_s.py finns stora delar av ett sådant program färdigt.
Vi har här infört följande abstrakta datatyper:

Abstrakt datatyp	**Beskrivning**
board	En ändlig avbildning med tre objekt av datatypen row som indexeras med talen 1-3.
row	En ändlig avbildning med tre objekt av datatypen piece.
piece	En primitiv datatyp som har något av värdena ‘empty’, ‘ring’ eller ‘cross’
position	En tupel med två siffervärden som beskriver en koordinat på spelplanen
posseq	En sekvens av objekt av datatypen position

I filen tictactoe_s.py finns primitiva funktioner för samtliga dessa datatyper. Där finns också en global variabel WINNING_POSITIONS som är en lista med objekt av datatypen posseq och som innehåller alla möjliga

sätt att vinna spelet.

Deluppgift A (2p)

Definiera funktionen make_move. Funktionen ska i en spelplan lägga in en pjäs på en given position. Kontroller behöver inte utföras. Funktionen ska givetvis använda sig av de definierade primitiverna.

Deluppgift B (3p)

Definiera funktionen is_winner. Funktionen ska givet en spelplan och en pjäs avgöra om vinst föreligger för pjäsen.

Uppgift 5

Följande skiss är en representation av en riktad graf med sex olika noder med beteckningarna A till F.

```
        -----> C
       /        \
A --> B <---     \
 \          \     \
  ---------> D <---
             \
             --> E --> F
```

Vi vill ha en funktion has_loop som kan kontrollera om man, genom att följa pilarna, kan hamna i en oändlig loop om man startar i en given nod. Vi representerar grafen som en dictionary där varje nods barn är en tupel. Ovanstående graf blir på detta sätt:

```
test_graph = {'a': ('b', 'd'), 'b': ('c'), 'c': ('d'), 'd': ('b', 'e'), 'e': ('f'), 'f': ()}
```

Vi vill att funktionen has_loop ska fungera så här:

```
>>> has_loop('a', test_graph)
True
>>> has_loop('c', test_graph)
True
>>> not has_loop('e', test_graph)
False
```

Om vi startar i någon av noderna A eller C och följer pilarna kommer vi ganska snart att komma tillbaka till noder vi redan har sett. Om vi däremot startar i noden E kommer vi inte att komma tillbaka någon gång.

Uppgift 6

Skriv en funktion give_change som beräknar vilken växel som ska lämnas tillbaka när man handlar kontant i en affär. Funktionen give_change ska som indata ta ett heltal som motsvarar den växel som ska lämnas tillbaka, och den ska returnera

en lista med olika valörer. Exempel:

```
>>> give_change(177)
[100, 50, 20, 5, 1, 1]
>>> give_change(206)
[100, 100, 5, 1]
```

De valörer på sedlar och mynt som finns tillgängliga får hårdkodas på ett ställe i koden. Lösningen ska så långt det är möjligt vara datadriven, d.v.s. onödig upprepning av programkod ska undvikas.